


Rust 并发编程

Concurrent Programming in Rust for Beginners

About Me

于海鑫 · brupst 

- 17 级 CS 本科生 @ BUPT
- Rust 萌新 
- GitHub: name1e5s
- 最近在搓乱序超标量处理器核，对计算机系统结构比较感兴趣

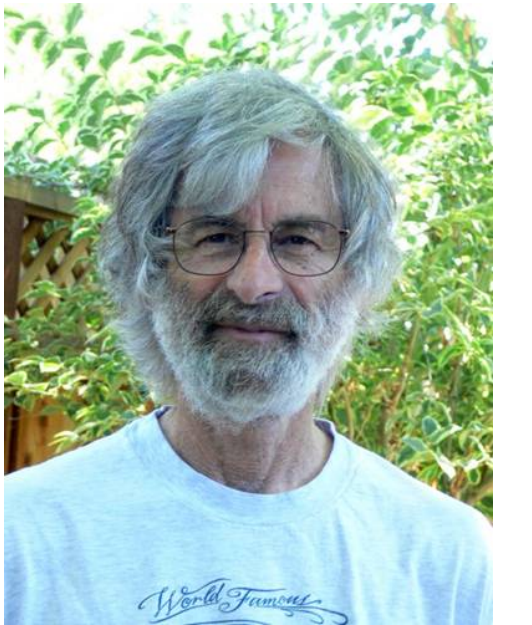
目录

- 什么是并发
- 并发的好处 & 问题
- 多线程模型
- 以及一点点异步编程

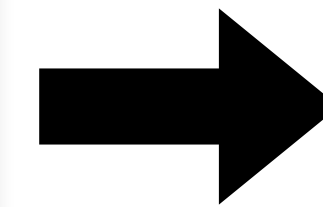
什么是并发

什么是并发

by Leslie Lamport



Definition. The relation “ \rightarrow ” on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same process, and a comes before b , then $a \rightarrow b$. (2) If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events a and b are said to be *concurrent* if $a \nrightarrow b$ and $b \nrightarrow a$.



independent

in Simple English

“Time, Clocks and the Ordering of Events in a Distributed System”, Leslie Lamport

什么是并发

把上面的定义类比到编程上...

“Programming as the **composition** of **independently** executing parts.”

什么是并发

并发 ≠ 并行



并发：同时处理多个任务



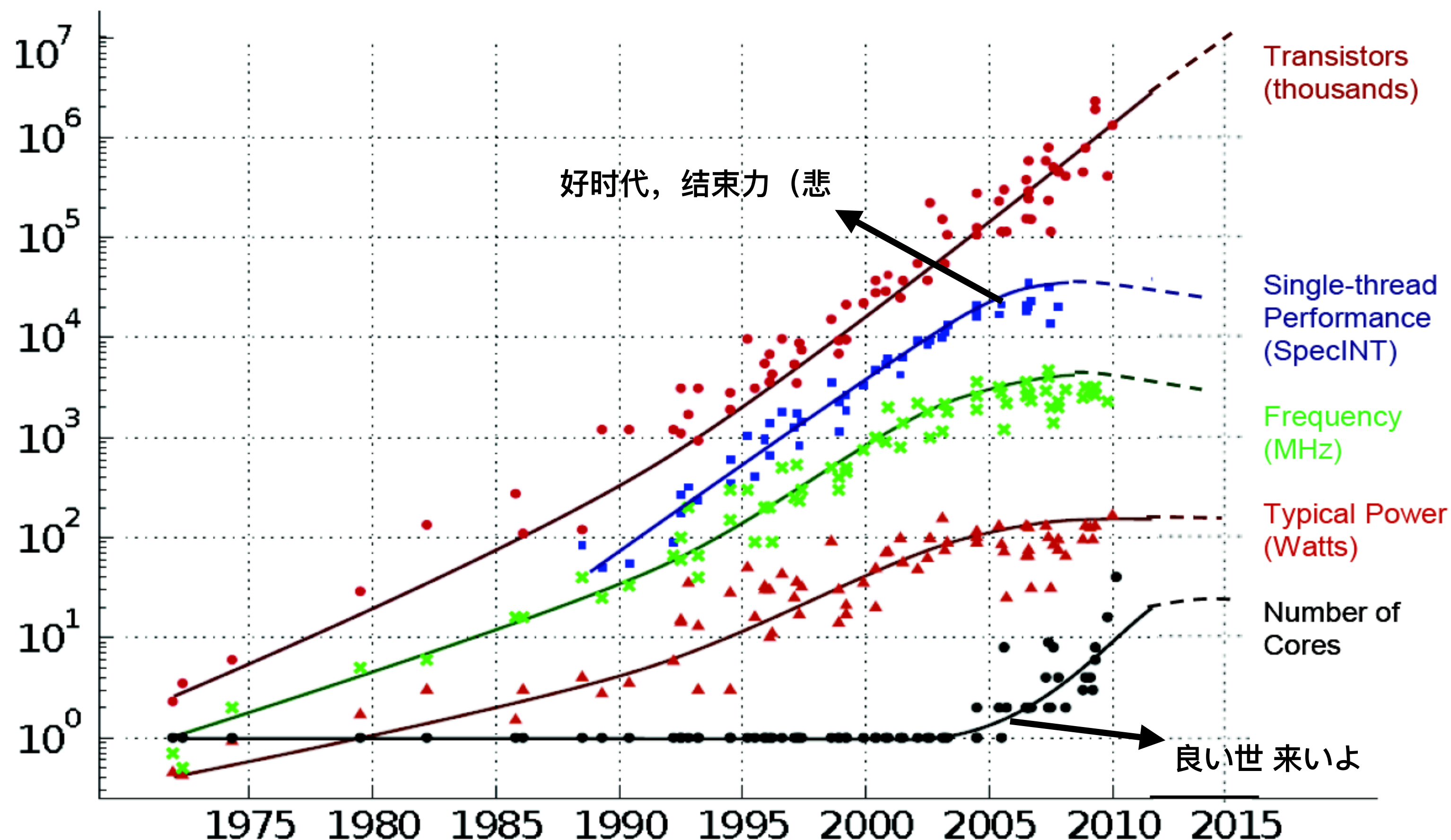
并行：同时执行多个任务

什么是并发

并发的优点

- (多核系统) 加速执行
- 可以将任务切成多个小单元并行执行
- 隐藏操作延迟
- CPU 可以在等待 I/O 时处理其他任务
- 增加吞吐量

35 YEARS OF MICROPROCESSOR TREND DATA

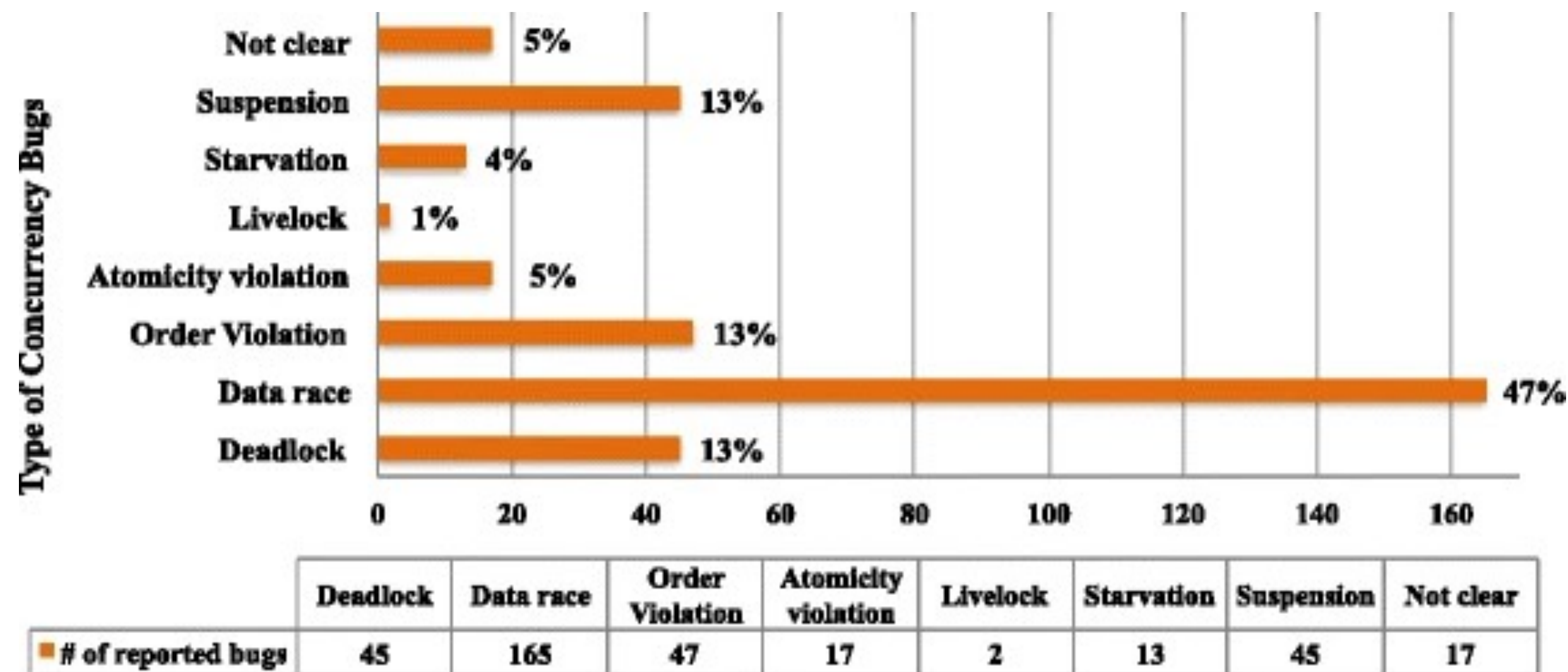


Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

什么是并发

并发引入的新问题

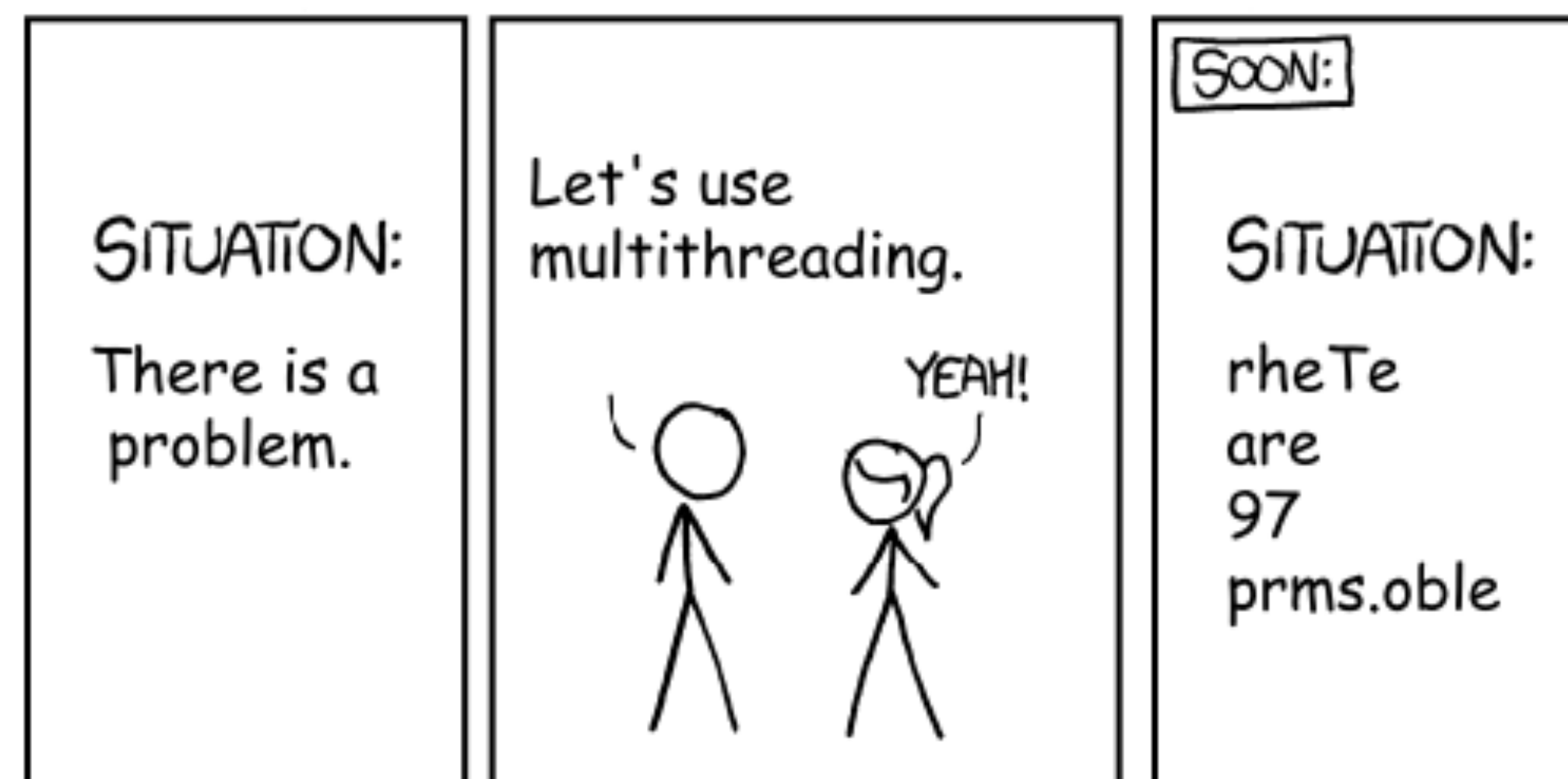
- 各个任务的执行顺序不确定，导致以非预期的顺序读写资源
- 各个任务互相等待资源（死锁）



什么是并发

数据竞争、竞态条件

- 数据竞争
 - 多个线程同时访问同一个内存地址
 - 这些线程中存在对于该内存地址的写操作
 - 这些线程之间不同步
- 竞态条件
 - 不同的任务执行顺序会造成不同的执行结果



什么是并发

并发编程可能遇到的问题一览

- 内存安全问题 ➡ 不搞并发这些问题也很常见

- 空指针/野指针

- 悬空指针

- 非法释放内存

- 等等

- 数据竞争

- 死锁

}
}
}
Sharing + Mutation

什么是并发

并发编程可能遇到的问题一览

- 内存安全问题 ➡ 不搞并发这些问题也很常见
 - 空指针/野指针
 - 悬空指针
 - 非法释放内存
 - 等等
- 数据竞争
- 死锁



can help!

Rust 中的并发

Rust 中的并发

多线程模型 for Beginners

“ In most current operating systems, an **executed program's code** is run in a **process**, and the operating system manages multiple processes at once. Within your program, you can also have **independent parts that run simultaneously**. The features that run these independent parts are called **threads**.”

The Rust Programming Language, Chap. 16 Fearless Concurrency

Rust 中的并发

多线程模型

- Language Thread : System Thread
 - M:N 模型: `libgreen`
 - 需要运行时支持
 - 不符合 Rust 设计目标, 已弃用👎
 - 1:1 模型: `std::thread`
 - 对操作系统提供的线程功能的包装

Rust 多线程模型

相关 API

- `std::thread` 线程的生成与管理
- `std::sync` 同步相关
 - `std::sync::atomic::fence` 内存屏障
 - `std::sync::atomic` 原子类型，可以用于构建更高级的同步原语
 - `std::sync::{Mutex, RwLock, Condvar}` 更高级的同步原语
 - `std::sync::mpsc` 用于线程间通信的队列

Rust 多线程模型

std::thread

```
1  use std::thread;
2  use std::time::Duration;
3
4  fn main() {
5      let handle = thread::spawn(|| {
6          for i in 1..10 {
7              println!("hi number {} from the spawned thread!", i);
8              thread::sleep(Duration::from_millis(1));
9          }
10     });
11
12     handle.join().unwrap();
13
14     for i in 1..5 {
15         println!("hi number {} from the main thread!", i);
16         thread::sleep(Duration::from_millis(1));
17     }
18 }
```

```
1  use std::thread;
2
3  thread::yield_now();
4  thread::park();
```

“By leveraging **ownership** and **type checking**, many concurrency errors are **compile-time errors** in Rust rather than **runtime errors**.”

*The Rust Programming Language, Chap. 16 **Fearless Concurrency***

Fearless Concurrency

Ownership

```
1  use std::thread;
2
3  fn main() {
4      let mut v = vec![1, 2, 3];
5      let handle = thread::spawn(move || {
6          v.push(114514);
7      });
8      drop(v);
9      handle.join().unwrap();
10 }
```

error[E0382]: use of moved value: `v`
→ src/main.rs:8:10

```
4 |     let mut v = vec![1, 2, 3];
   |     ----- move occurs because `v` has type `Vec<i32>`, which does not implement the `Copy` trait
5 |     let handle = thread::spawn(move || {
   |                                ----- value moved into closure here
6 |         v.push(114514);
   |         - variable moved due to use in closure
7 |     });
8 |     drop(v);
   |         ^ value used here after move
```

Fearless Concurrency

Type Checking

- `std::marker:: {Send, Sync}`
- 空 Trait, 仅作为标记
- 编译器推导, 大部分时间不需要手动实现

```
pub unsafe auto trait Send { }
```

```
pub unsafe auto trait Sync { }
```

Fearless Concurrency

Send: Thread-Safe **Exclusive** Access

- 实现了 Send 的类型，可以安全地在线程之间传递所有权 ➡ 跨线程移动
- 绝大多数类型都可以 Send
- 不实现 Send 的典型例子 `std::rc::Rc`
 - `Rc::clone` 会造成引用计数增加，该操作不是原子的
- 其对应类型 `std::sync::Arc` 使用原子计数，可以 Send

```
1 pub fn spawn<F, T>(f: F) -> JoinHandle<T>
2 where
3     F: FnOnce() -> T,
4     F: Send + 'static,
5     T: Send + 'static,
```


Fearless Concurrency

Sync: Thread-Safe **Shared** Access

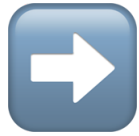
不可变借用 ≠ 不可以改变内部数据
内部可变性

- 实现了 Sync 的类型，可以安全地在线程之间传递**不可变借用** ➡ 跨线程共享
- 显然 $T: \text{Sync} \text{ iff } \&T: \text{Send}$
- 若 $T: \text{Sync}$ ，则
 - T 的全部 pub field 均 Sync
 - T **無** 内部可变性...
 - 或者 T 自己对多线程访问的安全**负责**

得自己动手标 Sync

Rust 多线程模型

`std::sync::atomic`

- 最底层的同步原语
- 大部分由 CPU 提供支持  `cmpxchg`
`ldxr/stxr`
`amoswap.w.aq/rl`
- 用于实现非常炫酷的无锁数据结构...
- 以及 Mutex 等更高级的同步原语

Rust 多线程模型

土法制 Mutex

```
1  use std::sync::atomic::AtomicBool;
2  use std::sync::atomic::fence;
3  use std::sync::atomic::Ordering;
4
5  pub struct Mutex {
6      flag: AtomicBool,
7  }
8  impl Mutex {
9      pub fn new() -> Mutex {
10         Mutex {
11             flag: AtomicBool::new(false),
12         }
13     }
14     pub fn lock(&self) {
15         // Wait until the old value is `false`.
16         while self.flag.compare_and_swap(false, true, Ordering::Relaxed) != false {}
17         // This fence synchronizes-with store in `unlock`.
18         fence(Ordering::Acquire);
19     }
20     pub fn unlock(&self) {
21         self.flag.store(false, Ordering::Release);
22     }
23 }
```

* https://en.cppreference.com/w/cpp/atomic/memory_order

Rust 多线程模型

`std::sync`

- Mutex 互斥锁
- RwLock 读写锁，多读取单写入
- Arc 原子引用计数
- Condvar 条件变量

Rust 多线程模型

std::sync::{Arc, Mutex}

- 有没有
Rc<RefCell<T>>
的即视感?

```
1  use std::sync::{Arc, Mutex};
2  use std::thread;
3
4  fn main() {
5      let counter = Arc::new(Mutex::new(0));
6      let mut handles = vec![];
7
8      for _ in 0..10 {
9          let counter = Arc::clone(&counter);
10         let handle = thread::spawn(move || {
11             let mut num = counter.lock().unwrap();
12             *num += 1;
13         });
14         handles.push(handle);
15     }
16
17     for handle in handles {
18         handle.join().unwrap();
19     }
20
21     println!("Result: {}", *counter.lock().unwrap());
22 }
```

Rust 多线程模型

std::sync::RwLock

- `unsafe impl<T: ?Sized + Send + Sync> Sync for RwLock<T> {}`
- `unsafe impl<T: ?Sized + Send> Sync for Mutex<T> {}`

```
1  use std::sync::RwLock;
2
3  let lock = RwLock::new(5);
4
5  // many reader locks can be held at once
6  {
7      let r1 = lock.read().unwrap();
8      let r2 = lock.read().unwrap();
9      assert_eq!(*r1, 5);
10     assert_eq!(*r2, 5);
11 } // read locks are dropped at this point
12
13 // only one write lock may be held, however
14 {
15     let mut w = lock.write().unwrap();
16     *w += 1;
17     assert_eq!(*w, 6);
18 } // write lock is dropped here
```

Rust 多线程模型

std::sync::mpsc

- 多生产者单消费者管道，用于线程间的消息传递

```
1 use std::sync::mpsc;
2 use std::thread;
3
4 fn main() {
5     let (tx, rx) = mpsc::channel();
6
7     thread::spawn(move || {
8         let val = String::from("hi");
9         tx.send(val).unwrap();
10    });
11
12    let received = rx.recv().unwrap();
13    println!("Got: {}", received);
14 }
```

“ Do not communicate by sharing memory;
instead, share memory by communicating.”

Rust 中的并发

异步编程

- 在较少的 OS 线程上面运行非常多的并发任务 ➡ 线程切换开销较大
- 使用 `async/.await` 语法以同步的方式写异步代码
 - `async` 将函数/代码块转换为异步的
 - `.await` 异步地等待异步函数/代码块返回结果

```
1 use async_std::task::spawn;
2 use async_std::net::{TcpStream, TcpListener};
3 use futures::{AsyncReadExt, AsyncWriteExt, StreamExt};
4
5 async fn handle_connection(mut stream: TcpStream) {
6     let mut buffer = [0; 1024];
7     stream.read(&mut buffer).await.unwrap();
8     //<-- snip to form a response -->
9     stream.write(response.as_bytes()).await.unwrap();
10    stream.flush().await.unwrap();
11 }
12
13 #[async_std::main]
14 async fn main() {
15     let listener = TcpListener::bind("127.0.0.1:7878").await.unwrap();
16     listener
17         .incoming()
18         .for_each_concurrent(None, |stream| async move {
19             let stream = stream.unwrap();
20             spawn(handle_connection(stream));
21         })
22         .await;
23 }
```

Rust 异步编程

内部实现

```
1  pub enum Poll<T> {
2      Ready(T),
3      Pending,
4  }
5
6  pub trait Future {
7      /// The type of value produced on completion.
8      #[stable(feature = "futures_api", since = "1.36.0")]
9      type Output;
10     #[lang = "poll"]
11     #[stable(feature = "futures_api", since = "1.36.0")]
12     fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
13 }
```



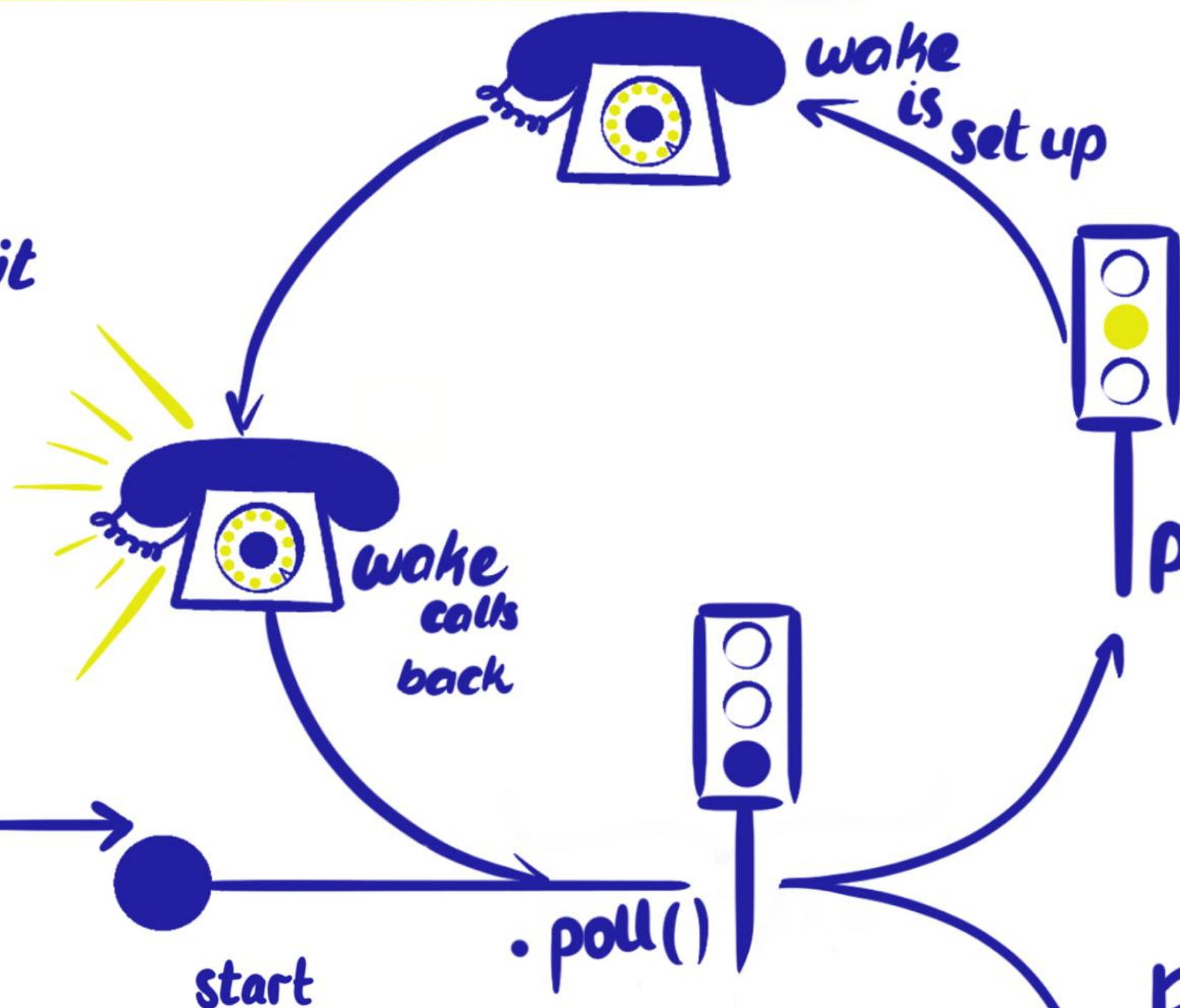

Futures

Rust's Async computation available as a Trait

`Std::Future`

Futures have 2 states represented by `Poll`

- i. Pending
- ii. Ready



Writing Futures by hand requires a few workarounds:

~ you are only able to pass around STATIC VALUES

~ to borrow: use `unsafe` or pass data by `Value`. That Value then gets returned back with `Result`.

Asynchronous code in synchronous fashion. → `SYNTAX`

But needs `block.on` to run in main

The `async/await` block of code is transformed into a STATE MACHINE

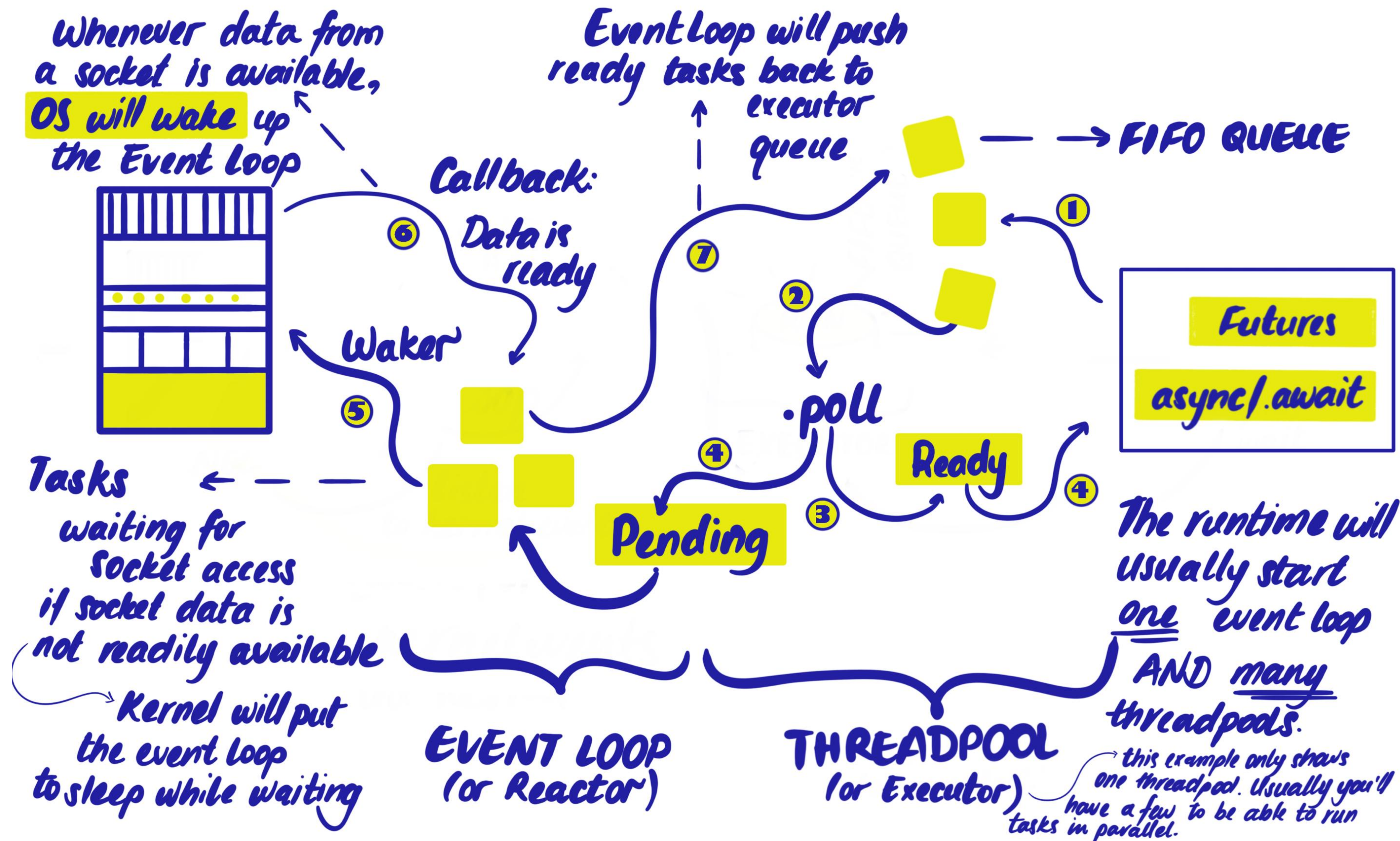
Futures will not execute unless they are spawned or polled

need an Executor + runtime

Solution:
Async/Await

`async/await`

Async Runtime



For more information:

<https://book.async.rs>

<https://rust-lang.github.io/async-book>

Thanks
for your patience

*The safest program is the
program that doesn't compile.*

~ ancient Rust proverb