第二十一讲：异步编程 (Asynchronous Programming)

第 1 节：Background

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 6 日

提纲

Ref:

- Futures Explained in 200 Lines of Rust, by Carl Fredrik Samson
- Writing an OS in Rust - Async/Await, by Philipp Oppermann
- Zero-cost futures in Rust, by Aaron Turon
- Rust' s Journey to Async/Await, by Steve Klabnik
- Asynchronous Programming in Rust

recap: Multitasking

Non-Preemptive multitasking

- The programmer ‘yielded‘ control to the OS
- Every bug could halt the entire system
- Example: Windows 95

Preemptive multitasking

- OS can stop the execution of a process, do something else, and switch back
- OS is responsible for scheduling tasks
- Example: UNIX, Linux

recap: User-level Thread

Advantages

- Simple to use
- A "context switch" is reasonably fast
- Each stack only gets a little memory
  - You can have hundreds of thousands of user-level threads running
- Easy to incorporate preemption

Drawbacks

- The stacks might need to grow
  - Solving this is not easy and will have a cost
- Need to save all the CPU state on every switch
- Complicated to implement correctly if you want to support many different platforms

Example: Green Threads

recap: Kernel-supported Threads

Advantages

- Easy to use
- Switching between tasks is reasonably fast
- Geting parallelism for free

Drawbacks

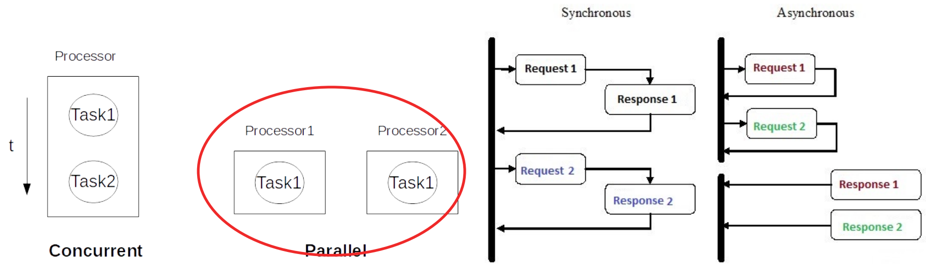
- OS level threads come with a rather large stack
- There are a lot of syscalls involved
- Might not be an option on some systems, such as http server

Example:

- Using OS threads in Rust

recap: What is async?

- Parallel: do multiple things at once
- Concurrent: do multiple things, not at once
- Asynchronous: Describe lang/prog features that enable parallelism & concurrency
- Task: Some computation running in a parallel or concurrent system



Why we need async?

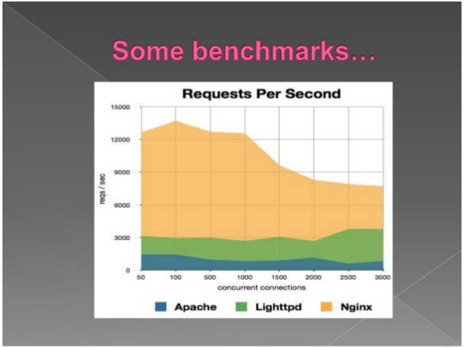
C10K Problem in 1999 ... C100K，C1M，C10M，C100M ⋯

- 网络服务在处理数以万计的客户端连接时，往往出现效率低下甚至完全瘫痪，这被称为 C10K 问题
- C10K 问题的提出者 Dan Kegel：软件工程师
- Web1.0 Ok！ Web2.0 Cry!

Why we need async?

C10K Problem in 1999 ... C100K，C1M，C10M，C100M ⋯

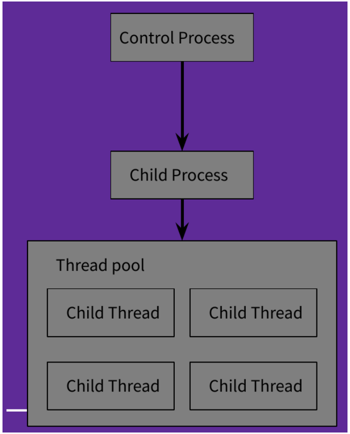
- 网络服务在处理数以万计的客户端连接时，往往出现效率低下甚至完全瘫痪，这被称为 C10K 问题
- C10K 问题的提出者 Dan Kegel：软件工程师
- Web1.0 Ok！ Web2.0 Cry！核心问题：**时间开销** + **空间开销**

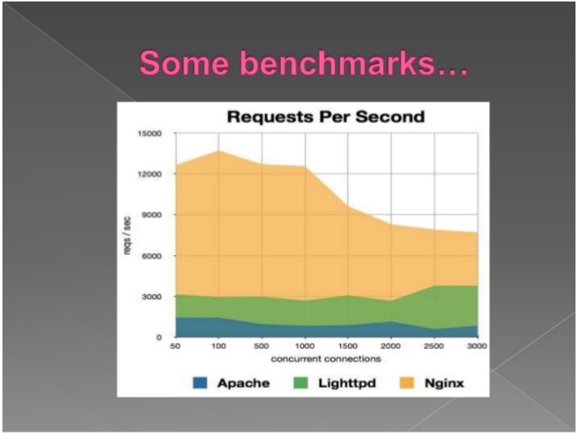


Why we need async?

解决方法: C10K Problem in 1999 … C100K，C1M，C10M，C100M ⋯

- Serve one client with each thread/process, and use blocking I/O : Apache、ftpd

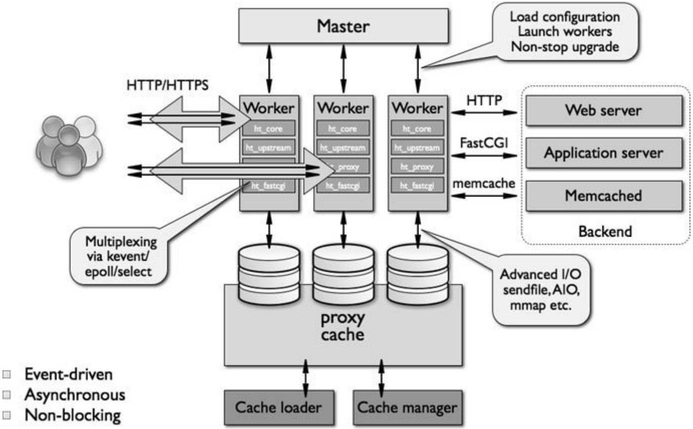




Why we need async?

解决方法: C10K Problem in 1999 ... C100K，C1M，C10M，C100M ⋯

- Serve many clients with each thread, and use asynchronous I/O : nginx

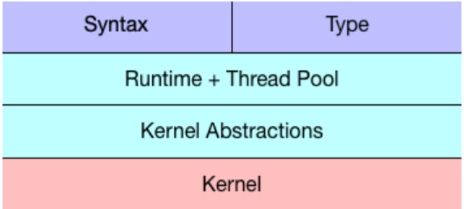


Why we need async?

解决方法: C10K Problem in 1999 ... C100K，C1M，C10M，C100M ···

- Serve many clients with each thread, and use asynchronous I/O : nginx

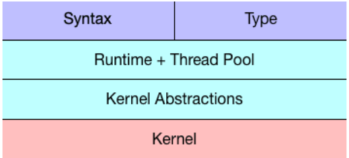
Who provide async mechanism?

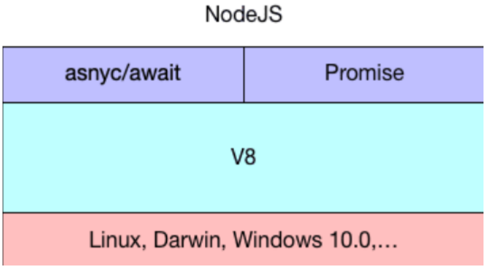


- Implement your own way of handling threads and queues on program level (green threads)
- Add syntactic sugar to your language so the runtime/compiler can identify async parts of the code
- Add async types so they can notify when they are "done"

F# added to the core design in 2007: computation expressions and their application to asynchronous programming

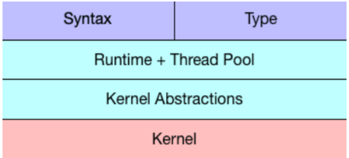
Who provide async mechanism?



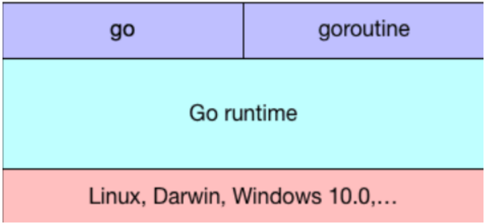



```
const async_method = async () => {
  const dbResults = await dbQuery();
  const results = await serviceCall(dbResults);
  console.log(results);
}
```

Who provide async mechanism?



Go

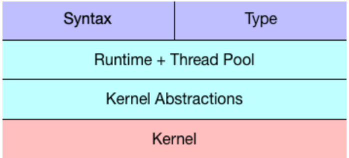



```
f(greeting string) {
    fmt.Println(greeting, ", World!")
}

go f("Hello")
```

Who provide async mechanism?

Rust




```
async fn hello_world() {
    let x: u8 = foo().await;
    println!("{} hello, world!",x);
}
fn main() {
    let future = hello_world(); // do nothing
    block_on(future); // print something
}
```

Async Prog: Callback based approaches

A callback based approach is to save a pointer to a set of instructions we want to run later together with whatever state is needed.

Advantages

- Easy to implement in most languages
- No context switching
- Relatively low memory overhead

Drawbacks

- Memory usage grows linearly with the number of callbacks
  - Each task must save the state it needs for later
- Callback hell: Hard to debug
- Require a substantial rewrite to go from a "normal" program flow to one that uses a "callback based" flow

Example: Callback based approaches

From callbacks to futures (deferred computation)

A callback based approach.

```
\\JavaScript
  setTimer(200, () => {
     setTimer(100, () => {
        setTimer(50, () => {
           console.log("I'm the last one");
        });
     });
  });
```

From callbacks to futures (deferred computation)

Promises: deal with the complexity which comes with a callback based approach.

```
\\JavaScript
    function timer(ms) {
    return new Promise((resolve) => setTimeout(resolve, ms));
    }

    timer(200)
    .then(() => return timer(100))
    .then(() => return timer(50))
    .then(() => console.log("I'm the last one"));
```

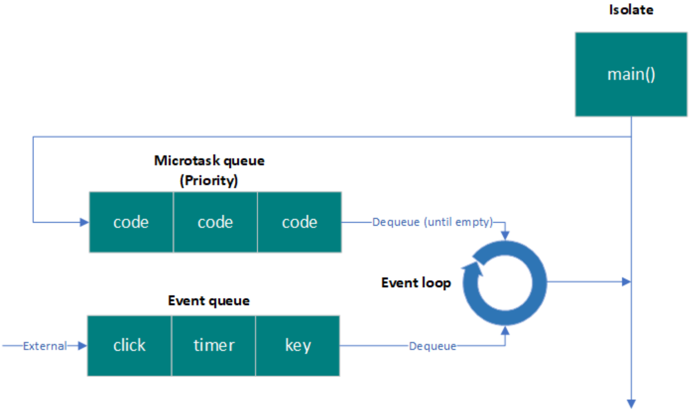
From callbacks to futures (deferred computation)

Promises: deal with the complexity which comes with a callback based approach.

```
\\javascript
  async function run() {
      await timer(200);
      await timer(100);
      await timer(50);
      console.log("I'm the last one");
  }
```

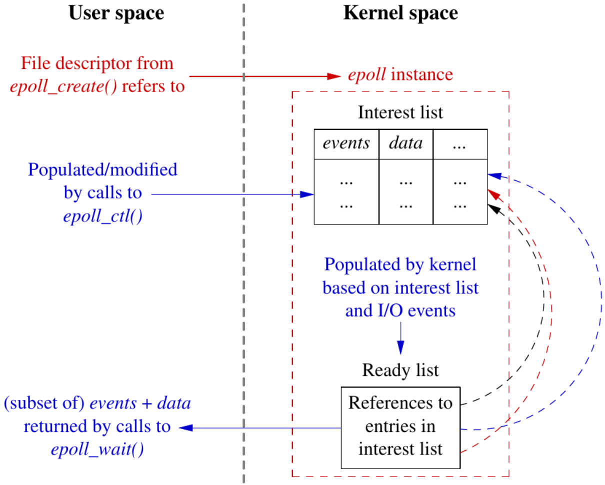
- The ‘run‘ function as a *pausable* task consisting of several sub-tasks
  - On each ”await” point it yields control to the scheduler
- When the sub-tasks changes state to either ‘fulfilled‘ or ‘rejected‘, the task is scheduled to continue to the next step

Event queue: Epoll, Kqueue and IOCP



- Epoll is the Linux way of implementing an event queue
- Kqueue is the MacOS way of implementing an event queue
- IOCP or Input Output Completion Ports is the way Window handles event queue

Epoll


User space
Kernel space
File descriptor from
epoll_create() refers to
epoll instance
Interest list
events
data
...
Populated/modified
by calls to
epoll_ctl()
Populated by kernel
based on interest list
and I/O events
Ready list
References to
entries in
interest list
(subset of) events + data
returned by calls to
epoll_wait()

Procedure for read data from a socket using epoll

1. Create an event queue by calling the syscall ‘epoll_create‘ or ‘kqueue‘
2. Ask the OS for a file descriptor representing a network socket
3. Register an interest in ‘Read‘ events on this socket
   - In order to receive a notification when the event is ready in the event queue we created
4. Call ‘epoll_wait‘ or ‘kevent‘ to wait for an event
   - Block (suspend) the thread it’s called on
5. When the event is ready, our thread is resumed, and return from our ”wait” call with data about the event
6. Call ‘read‘ on the socket we created

Example

- Epoll example
- Complete example