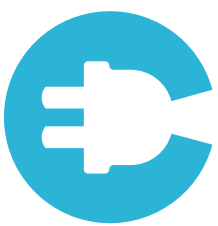
Async IO, Completed

In case of `compio`, an async runtime for Rust



Pop (@George-Miao) on behalf of `compio-rs`

https://github.com/compio-rs/compio

Authors

@Berrysoft Yuyi Wang

PhD candidate
Department of Engineering Physics
Tsinghua University

@George-Miao

Master's Student
Department of Computer Science
University of Maryland

Introduction to async Rust

A blocking example (pseudo code):

```
fn example() {
  let socket = TcpListener::bind("0.0.0.0:80")?;
  let mut buffer = [0; 512];

  while let Ok((stream, addr)) = socket.accept() {
      stream.read(&mut buffer)?;
      // ..
  }
}
```

`.accept()` and `.read()` blocks current thread.

Now turn it into an async one:

```
async fn example() {
  let socket = TcpListener::bind("0.0.0.0:80").await?;
  let mut buffer = [0; 512];

  while let Ok((stream, addr)) = socket.accept().await {
      stream.read(&mut buffer).await?;
      // ..
  }
  Ok(())
}
```

`.accept()` and `.read()` won't block, but still runs sequentially.

Async with concurrency:

```
async fn example() {
  let socket = TcpListener::bind("0.0.0.0:80").await?;

  while let Ok((stream, addr)) = socket.accept().await {
      spawn(async move {
          let mut buffer = [0; 512];
          stream.read(&mut buffer).await?;
          // ..
      });
  }
}
```

`accept()` and `read()` happens simultaneously, increase throughput.

Under the hood

Async rust builds around `trait Future`.

```
trait Future {
  type Output;

  fn poll(self: Pin<&mut Self>, ctx: &mut Context) -> Poll<Self::Output>;
}

enum Poll<T> {
  Ready(T),
  Pending,
}
```

- Represents “the operation will complete (`Ready`) sometime in future”
- If not ready, return `Pending` and obtain a `Waker` from `Context` to notify later
  - Effectively gives up the thread (yield) until being polled again
- Compiler generates a `Future` type (state machine) for each `async fn`

Runtime = IO + Executor

Reactor/Proactor interacts with the world, executor bridges them to rust

Runtime pt. 1, The Executor

`Futures` progress only when they're polled. A simple executor polls the future in a loop:

```
pub fn block_on<F: Future>(f: F) -> F::Output {
  let waker = /* create a waker */;
  let mut ctx = Context::from_waker(&waker);
  let mut f = std::pin::pin!(f);
  loop {
    if let Poll::Ready(result) = f.as_mut().poll(&mut ctx) {
      return result;
    }
    // poll the io
    // ... and that's why it is hard to mix different runtimes.
  }
}
```

Poll-Based (Reactor)

- Register a file descriptor with the kernel
- Poll = Check for readiness notification
- Perform the blocking I/O syscall knowing it won't block
- Examples: `epoll`/`kqueue`/`poll`

Completion-Based (Proactor)

- Submits the I/O request to the kernel
- Poll = Check for result
- Hand out ownership of buffers to the kernel
- Examples: `IOCP`/`io_uring`

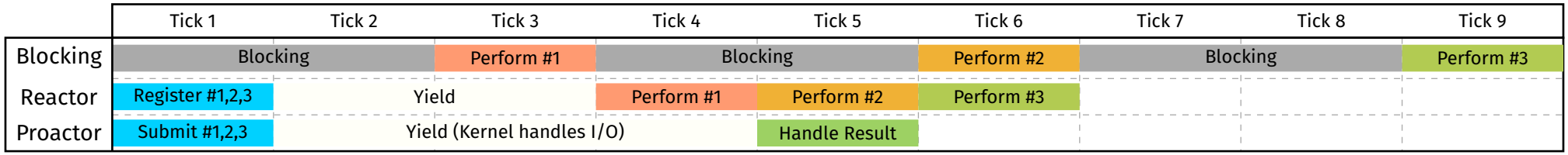
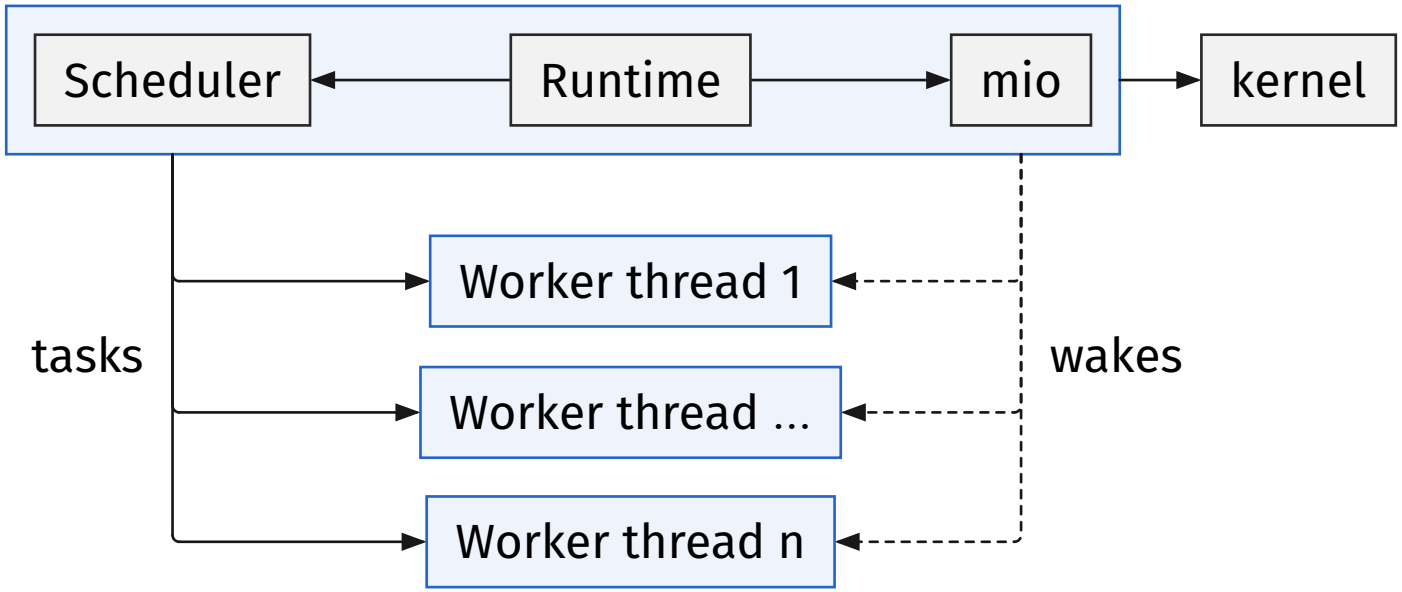


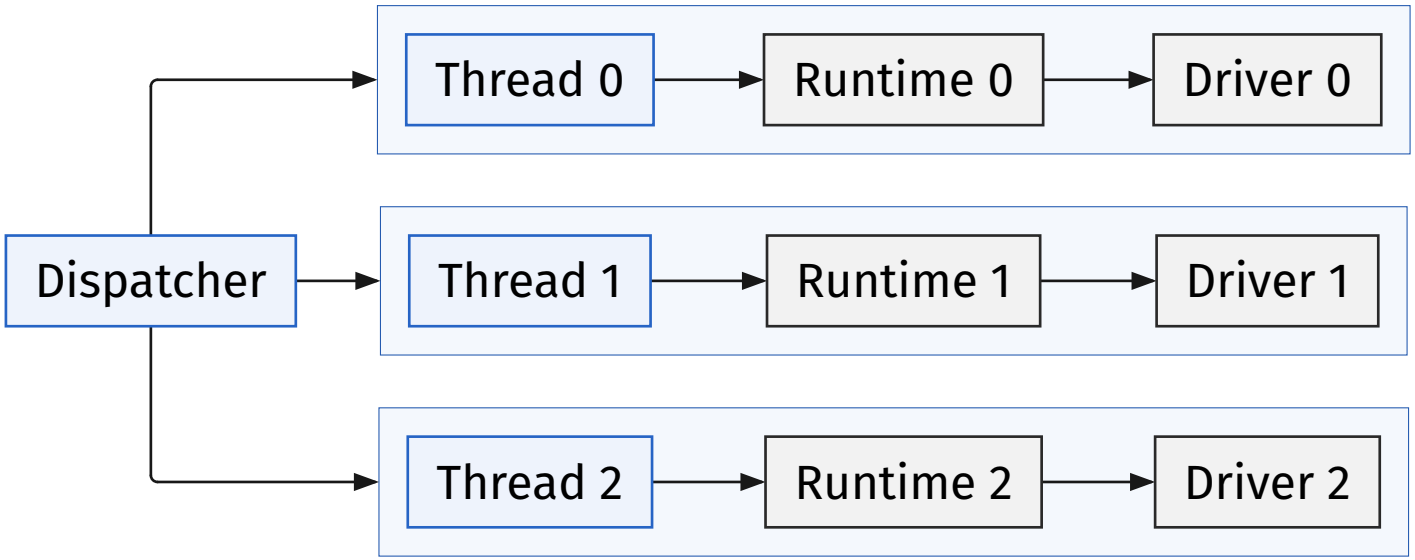
Figure 1: An over simplified comparison between blocking, poll-based, and completion-based IO

Go-to option: Tokio



- Poll-based async runtime (uses `mio`, a reactor library)
- Uses work-stealing scheduler to balance load across threads by default
  - Requires `Send & Sync` when `spawn`
  - Force using of synchronization (lock, atomic etc), reduces locality and predictability

Introducing Compio



- Completion-based async runtime
- Mostly local or share-nothing
- Low contention; locality; predictable
- Optional global dispatcher for load balancing

Compio Driver

Submit once; kernel reports completion with results.

```
let mut proactor = Proactor::new()?;
proactor.attach(socket)?;                    // required by IOCP on Windows
let op = Recv::new(socket, buf, flags);      // submit operation
let (res, op) = match proactor.push(op) {
  PushEntry::Ready(res) => res,              // completes immediately
  PushEntry::Pending(mut key) => loop {      // the executor loop
      proactor.poll(None)?;                  // poll the kernel
      match proactor.pop(key) {              // retrieve the result
      PushEntry::Ready(res) => break res,
      PushEntry::Pending(k) => key = k,
    }
  }
}?;                                          // `?` is available in nightly Rust
```

Cancellation safety

- In poll-based, cancel by doing nothing (the operation hasn't started)
- In completion-based, cancellation is asynchronous (tells kernel to stop)
- The operation may still complete after being cancelled

```
let mut server = TcpListener::bind("localhost:0").await.unwrap();
let task = server.accept();
let timeout = sleep(Duration::from_secs(5));
select! {
  client = task => {
    // Handle the client connection...
  }
  _ = timeout => {
    // What if there's still a client coming?
  }
}
```

Always forgettable

- Normally, `Future` in compio trys to clean up the operation when being dropped.
- But `Drop` is not reliable since forget (leak) is safe in Rust
- A reactor only accesses the buffer synchronously. No Problem.
- A proactor submit the buffer to the kernel
  - Causes Use-After-Free
- Current solution: pass the ownership:

```rust
let (n, buf) = sock.read(buf).await?;
```

```rust
let mut buffer = [0u8; 1024];
let fut = socket.read(&mut buffer);
let cx = /* dymmy Context */;
pin!(fut).poll(cx); // buffer submitted
forget(fut);        // No drop, no cancel
drop(buffer);       // It compiles! UAF!
```

Conclusion

- Asynchronous Rust provides highly customizable mechanism for parallel IO.
- Based on proactor, `compio` provides a completion-based async runtime, and makes high concurrency more efficient.
- Completion-based asynchronous IO requires evolution of Rust language.