WILL IT BLOCK? — 2024-02-07

Will this function block?

```
fn work(x: u64, y: u64) -> u64 {
    x + y
}
```

How about this function?

```
fn work(x: u32) -> u32 {
    let mut prev = 0;
    let mut curr = 1;
    for _ in 0..x {
        let next = prev + curr;
        prev = curr;
        curr = next;
    }
    curr
}
```

Or this one?

```
use sha256::digest;

fn work(input: &str) -> String {
    digest(input)
}
```

Will this function block?

```
use std::{thread, time::Duration};
fn work() {
    thread::sleep(Duration::from_nanos(10));
}
```

Perhaps this one will?

```rust
use std::{thread, time::Duration};
fn work() {
    thread::sleep(Duration::from_millis(10));
}
```

Or how about this one, does this block?

```rust
use std::{thread, time::Duration};
fn work() {
    thread::sleep(Duration::from_secs(10));
}
```

Let's try another; does this block?

```rust
use std::{thread, time::Duration};
fn work() {
    thread::spawn(|| {}).join();
}
```

Up next: does this block?

```rust
use std::{thread, time::Duration};
fn work() {
    println!("meow");
}
```

Last one; does this block?

```rust
use std::fs::File;
use std::io::prelude::*;
use std::os::unix::io::FromRawFd;

fn work() {
    let mut f = unsafe { File::from_raw_fd(1) };
    write!(&mut f, "meow").unwrap();
}
```

And so on.

Determining whether something "is blocking" is a messy endeavor because computers are messy. Whether something blocks is not something we can objectively determine, but depends on our own definitions and uses. A system call which resolves quickly might not be considered "blocking" at all [1], while a computation which takes a long time might be. The Tokio folks wrote the following about it a few years ago:

> *[...] it is very hard to define "progress". A naive definition of progress is whether or not a task has been scheduled for over some unit of time. For example, if a worker has been stuck scheduling the same task for more than 100ms, then that worker is flagged as blocked and a new thread is spawned. In this definition, how does one detect scenarios where spawning a new thread reduces throughput? This can happen when the scheduler is generally under load and adding threads would make the situation much worse.*

Even if we decide something is "blocking" after an arbitrary cutoff - that might still not do us any good if the overall throughput of the system is reduced. Determining whether something is objectively blocking or not might be the most useful question to be asking. Alice Ryhl (also of Tokio) had the following to say:

> *In case you forgot, here's the main thing you need to remember: Async code should never spend a long time without reaching an `.await`.*

I believe this is a much more useful framing. Rather than attempting (and failing) to categorize calls as "blocking", we should identify where we're spending longer stretches of time without yielding back to the runtime - and ensure we can correct it when we don't. In async-std I introduced the `task::yield_now` function to help with this [2]. Tokio subsequently adopted it [3], and for WASI's async model a similar function is being considered too.

The goal of this post is to show by example the ambiguity present when attempting to create an objective classification of which code counts as "blocking". As well as taking the opportunity to highlight previous writing on the topic.

Special thanks to Jim Blandy, who during my review of The Crab Book convinced me it is actually fine to use a synchronous mutex in asynchronous code, in turn pointing me towards Alice's writing.

NOTES

1. Modern storage and networking devices can get pretty darn fast. A modern PCIe connection can transfer 128GB/s or 1Tb/s. Sub-millisecond response times for storage are not unheard of either. And even network calls, for example between processes (containers) or between machines in a data center, can resolve incredibly quickly. ←

2. The original issue where I introduced this idea is still available in case anyone is reading about the motivation for this at the time. We very clearly modeled it after `thread::yield_now`, but async. ←
3. I believe they've been working on an alternative system to this for a few years now, but I'm not sure what the state of that is. ←

REFERENCES

▼ View all references

1. https://blog.yoshuawuyts.com/what-is-blocking/#read-speeds
2. https://tokio.rs/blog/2020-04-preemption
3. https://ryhl.io/blog/async-what-is-blocking/
4. https://docs.rs/async-std/latest/async_std/task/fn.yield_now.html
5. https://blog.yoshuawuyts.com/what-is-blocking/#post
6. https://docs.rs/tokio/latest/tokio/task/fn.yield_now.html
7. https://blog.yoshuawuyts.com/what-is-blocking/#new-tokio
8. https://github.com/async-rs/async-std/issues/290
9. https://www.red-bean.com/jimb/
10. https://www.oreilly.com/library/view/programming-rust-2nd/9781492052586/
11. https://ryhl.io/blog/async-what-is-blocking/