Rust async tasks, semaphores, timeouts and more

Rust has great async support. With a combination of the tokio runtime and its tools you can create async code that's easy to understand and full of features that are common in async/parallel world.

Some of those tools I'm talking about are:

- semaphores
- timeouts
- progress bars

Requirements

Tokio is the only requirement for this project so you can put that requirement into `Cargo.toml`. By default Tokio comes very "skinny" - all the features must be enabled. That can be done by enabling the `full` feature:

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

or enumerate only the needed ones:

```
tokio = { version = "1", features = ["sync", "rt", "macros",
↪  "rt-multi-thread", "time"]}
```

Task pool

The idea of a task/thread pool is very simple. You have a certain amount of available "runners" that you utilize. Let's say you have 50 URLs that you need to load (fetch HTML). You can:

- go one by one - slowest, no parallelism
- go in batches (like 10) at a time - faster, safest
- go all at once - fastest (probably), not safe, dangerous

By safe, I mean the following. If you query any other service, you don't want to flood it. You want to go "with big buckets you can carry as few times as possible". That means you are looking for the optimal balance between "grab all in one take" and "go in batches of 2" (smallest possible batches).

If you flood a service on the other end, these scenarios can happen:

- the service will rate-limit you
- the service will (temporarily) ban you
- you can "choke" the service and eventually kill it

Either of those approaches can result in getting what you need even slower or not getting anything at all.

So the best solution is to determine a batch of, let's say, 8. In our example we will go with batches of 2 just to demonstrate what's going on.

Once you have the pool size decided, you know what to do and how to do so. Here is the code.

```
use std::io;
use std::sync::Arc;

use tokio::sync::Semaphore;
use tokio::time::Duration;
use tokio::time::sleep;

#[tokio::main]
async fn main() -> io::Result<()> {
    // Configuration.
    const TASKS: u8 = 8;
    const TASKS_POOL: usize = 2;

    // Setup of variables.
    let sem = Arc::new(Semaphore::new(TASKS_POOL));
    let mut task_handles = vec![];

    // Spawn tasks
    for i in 0..TASKS {
        let task_sem = sem.clone();

        task_handles.push(tokio::spawn(async move {
            // Acquire a semaphore permit asynchronously.
            let _permit = task_sem.acquire().await.unwrap();

            println!("working in task {} ...", i);
            sleep(Duration::from_secs(1)).await;
        }));
    }

    // Join all task handles.
    for h in task_handles {
        h.await.unwrap();
    }

    Ok(())
```

```
}
```

A very important thing here is to know how Semaphore handles permits. Once a permit is given (by the `acquire()` method) semaphore waits till the permit is dropped. That means the variable holding the permit goes out of scope. A common mistake is to store the permit in _ variable. _ variable is a throwaway binding which means Rust immediately drops it. That results in the permit being given and immediately released for next tasks - no semaphore functionality and all tasks run at once.

```
// This ...
let _ = task_sem.acquire().await.unwrap();
// ... equals to this:
task_sem.acquire().await.unwrap();
drop(permit);
println!("working...");
```

JoinSet instead of handles

Joining task handles can be improved with JoinSet provided by Tokio. The biggest difference in comparison to the vector of task handles is that the vector preserves order task spawn order. JoinSet doesn't care when the task was spawned instead of that it's focused on task finish order - when task has finished.

You can also process tasks results as they come in - no need to wait for all tasks to finish first. That's what join_next() method is for. If you need to wait for all tasks to finish first you can use join_all() method. Here are both approaches in a code:

```
use std::io;
use std::sync::Arc;

use tokio::sync::Semaphore;
use tokio::task::JoinSet;
use tokio::time::Duration;
use tokio::time::sleep;

#[tokio::main]
async fn main() -> io::Result<()> {
    // Configuration.
    const TASKS: u8 = 8;
    const TASKS_POOL: usize = 2;

    // Setup of variables.
    let sem = Arc::new(Semaphore::new(TASKS_POOL));
```

```
    let mut task_set = JoinSet::new();

    // Spawn tasks
    for i in 0..TASKS {
        let task_sem = sem.clone();

        task_set.spawn(async move {
            // Acquire semaphore permit.
            let _permit = task_sem.acquire().await.unwrap();

            println!("working in task {} ...", i);
            sleep(Duration::from_secs(1)).await;

            i
        });
    }

    // Join all task handles.
    // println!("Results: {:#?}", task_set.join_all().await);

    // Join next - returns results as tasks complete.
    println!("Results:");
    while let Some(r) = task_set.join_next().await {
        println!("{:?}", r)
    }

    Ok(())
}
```

Timeouts

What if one of your tasks takes too long or gets stuck for some reason? That means all the handles won't get joined and your code cannot continue - it will hang (forever).

This is the time when it's a good idea to add timeout. Just like web servers have timeouts (30 seconds by default) tasks should have one too. The magic is done by wrapping the task execution code with timeout() method.

In the following example the task sleeps for the amount of seconds equal to its ID (`i` variable). So first tasks finish faster than later ones. The threshold is set to 5.5 seconds which makes the last 2 tasks timeout (they got canceled by `timeout()` function).

```
use std::io;
```

```rust
use std::sync::Arc;

use tokio::sync::Semaphore;
use tokio::task::JoinSet;
use tokio::time::Duration;
use tokio::time::sleep;
use tokio::time::timeout;

#[tokio::main]
async fn main() -> io::Result<()> {
    // Configuration.
    const TASKS: u8 = 8;
    const TASKS_POOL: usize = 2;

    // Setup of variables.
    let sem = Arc::new(Semaphore::new(TASKS_POOL));
    // Task set needs to have a type specified because later on we change
    // error type of nested Result from &str to String.
    let mut task_set: JoinSet<Result<u8, Result<u8, String>>> =
     ↪  JoinSet::new();

    // Spawn tasks
    for i in 0..TASKS {
        let task_sem = sem.clone();

        task_set.spawn(async move {
            // Acquire semaphore permit.
            let _permit = task_sem.acquire().await.unwrap();

            timeout(Duration::from_millis(5500), async {
                // Here begins the task core.
                println!("working in task {} ...", i);
                // Let's sleep for the "i" amount of seconds.
                sleep(Duration::from_secs(i.into())).await;

                i
            })
            .await
            .map_err(|_e| Err(format!("timed out: {}", i)))
        });
    }

    // Join all task handles.
```

```
    println!("Results: {:#?}", task_set.join_all().await);

    Ok(())
}
```

Progress bar

The last thing one might add to this post example code is a progress bar. It's always polite to notify a user how the execution is going so the user can estimate how long will it take to finish and how fast it's advancing.

The progress bar needs to be updated once any task finishes so here we switch to `join_next()` approach:

```
use std::io::{self, Write};
use std::sync::Arc;

use tokio::sync::Semaphore;
use tokio::task::JoinSet;
use tokio::time::Duration;
use tokio::time::sleep;
use tokio::time::timeout;

#[tokio::main]
async fn main() -> io::Result<()> {
    // Configuration.
    const TASKS: u8 = 8;
    const TASKS_POOL: usize = 2;

    // Setup of variables.
    let sem = Arc::new(Semaphore::new(TASKS_POOL));
    // Task set needs to have a type specified because later on we change
    // error type of nested Result from &str to String.
    let mut task_set: JoinSet<Result<u8, Result<u8, String>>> =
     ↪  JoinSet::new();

    // Spawn tasks
    for i in 0..TASKS {
        let task_sem = sem.clone();

        task_set.spawn(async move {
            // Acquire blocking semaphore permit.
```

```
            let _permit = task_sem.acquire().await.unwrap();

            timeout(Duration::from_millis(5500), async {
                // Let's sleep for the "i" amount of seconds.
                sleep(Duration::from_secs(i.into())).await;

                i
            })
            .await
            .map_err(|_e| Err(format!("timed out: {}", i)))
        });
    }

    // Join all task handles.
    print!("Progress: 0%");
    let mut i = 0;

    while let Some(_r) = task_set.join_next().await {
        i += 1;
        // Print out percentage completion.
        print!(" ... {:.0}%", 100.0 / (TASKS as f32 / i as f32));
        io::stdout().flush().unwrap();
    }
    println!();

    Ok(())
}
```

Very important thing here is to “flush” output after each progress bar update so user can see the update immediately. This prevents buffering.

When to use

This model should be used in cases when you:

- can accidentally flood the other end
- the other end can deny your tasks
- the other end policy can change anytime
- you want to stay on the safer side