Async Rust and 9p server

Wedson Almeida Filho

Agenda

Introduction

Motivation, async Rust, kernel implementations

9p server

Description, some implementation samples

Demo

Discussion

Motivation

Showcase productivity gains from using Rust

- In addition to security benefits

Considerable attack surface

- For example, receiving untrusted data over the network

Pure software module

- No inherent unsafety due to bus mastering devices

Started looking at `ksmbd`

- Not an ideal initial project because of complexity of the protocol
- Also requires a user-space component

Async Rust

Talked about it at OSS North America: link

In summary:

- Compiler automatically creates a state machine from thread-like code
- Kernel crate implements executors and reactors

Workqueue Executor

Spawning tasks

Allocates task: contains future plus executor-specific state (e.g., `work_struct`)

Adds to task list

Wakes task up

Waking tasks up

Enqueues task for running (e.g., `queue_work_on`)

On worker thread: accesses revocable task, poll future, cleans it up when it completes

Tearing down

All state is dropped (more on this later)

Socket Reactor

Initialisation

- Pinned larger struct containing some state plus wait queue entry (`wait_queue_entry`)
- Wait queue entry with a custom function (`init_waitqueue_func_entry`)
- Adds entry to the socket's wait queue (`add_wait_queue`)

Waking up

- Wait queue callback is called: uses `container_of` to get to outer struct
- Checks mask for filter callbacks (`EPOLLIN`, `EPOLLOUT`, etc)
- Calls `Waker::wake` to instruct executor to run task again

Cleaning up

- Removes entry from socket's wait queue (`remove_wait_queue`)

9p file server

What is 9p?

Plan 9 is an operating system from Bell Labs (link)

- Originally designed by Ken Thompson, Rob Pike, et al.

Included a remote file system protocol: Plan 9 File Protocol, 9P

- The Linux kernel already includes a 9p client
- Qemu implements a server to share a host directory with guest

Straightforward: original protocol only includes 10 operations

What is implemented

Not a file system

Though we have some support for it here

Exposes the local file system over the network

Read-only for now

Implements `9P2000.L` – Linux extensions

WIP but available here

Code stats

```
fs/k9pd/Kconfig     |  10 +
fs/k9pd/Makefile    |   5 +
fs/k9pd/buffer.rs   | 286 +++++++++++++++
fs/k9pd/k9pd.rs     | 171 +++++++++
fs/k9pd/localfs.rs  | 382 +++++++++++++++++++++
fs/k9pd/protocol.rs | 200 +++++++++++
6 files changed, 1054 insertions(+)
```

Receiving requests

```
async fn next_pdu(&self, max_size: u32) -> Result<Box<[u8]>> {
   // Read the length.
   let mut len_in_bytes = [0u8; 4];
   self.stream.read_all(&mut len_in_bytes).await?;

   let len = u32::from_le_bytes(len_in_bytes).checked_sub(4).ok_or(EIO)?;
   if len > max_size {
       return Err(E2BIG);
   }

   // Allocate the buffer and read the rest.
   self.stream.alloc_read_exact(len as usize).await
}
```

Dispatching requests

```
loop {
   let pdu = conn.next_pdu(max_size).await?;
   let (_op, tag) = protocol::get_op_tag(&pdu)?;
   let res = spawn_task!(
      executor.as_ref_borrow(), conn.clone().handle_pdu(tag, pdu));
   if let Err(e) = res {
       conn.write_result(|b| protocol::error(b, tag, e)).await?;
   }
}
```

Serialising writes

```
async fn write(&self, buf: &[u8]) {
   let mut inner = self.inner.lock().await;
   if inner.err.is_err() {
       // A previous write failed so we won't even try this one.
       return;
   }
   let ret = self.stream.write_all(buf).await;
   if ret.is_err() {
       // Store away the error.
       inner.err = ret;
   }
}
```

Cleaning up

Executor keeps track of all incomplete tasks

Auto-stop handles stop executors when they go out of scope

Stopping an executor waits for running tasks to go to sleep

- Also ensures that sleeping tasks don't wake up anymore
- Drops all tasks

"Local variables" of async functions are also dropped

- Reactors unregister from subsystems
- Allocations are freed, ref-counted objects are released, etc.

Stopping an executor results in everything being dropped automatically

Demo

Discussion

Possible topics

Additional potential candidates to be written in async Rust

- Driver state machines?

Additional reactors

- kiocb, urb, bio?

Implementing more non-blocking operations/primitives

- Select
- Communication channels (send/receive data structures)
- Memory allocation (GFP_KERNEL): can we give up the worker instead of sleeping?
- Read-write mutexes
- Condition variables

Possible topics (cont'd)

Integration with user-space implementations

- Tokio panics on memory allocation failures

Destruction of mutual exclusion primitives

- State may be inconsistent
- Release the lock? Keep it locked? Poising (which implies acquisition may fail)?

Fault isolation

- Should we unwind stacks, drop objects, etc. for panics in tasks?
- At the moment, we kill a workqueue thread

Thank you!