Introduction to thu-learn-fuse

(The FUSE Project Report)

MashPlant

June 3, 2020

Table of Contents

Contents

Web Learning API

Inspired by https://github.com/Harry-Chen/thu-learn-lib.

Web Learning API

Inspired by https://github.com/Harry-Chen/thu-learn-lib.

TypeScript

```typescript
public async getSemesterIdList(): Promise<string[]> {
  const response = await this.#myFetch(URL.
      LEARN_SEMESTER_LIST());
  const semesters = (await response.json()) as string[];
  // sometimes web learning returns null, so confusing...
  return semesters.filter((s) => s != null);
}
```

Rust

```rust
pub async fn semester_id_list(&self) -> Result<Vec<Id>> {
  let res = self.0.get(SEMESTER_LIST).send().await?.json::<
      Vec<Option<String>>>().await?;
  Ok(res.into_iter().filter_map(|x| x).collect())
}
```

Web Learning API

Inspired by https://github.com/Harry-Chen/thu-learn-lib.

TypeScript

```typescript
public async getSemesterIdList(): Promise<string[]> {
  const response = await this.#myFetch(URL.
      LEARN_SEMESTER_LIST());
  const semesters = (await response.json()) as string[];
  // sometimes web learning returns null, so confusing...
  return semesters.filter((s) => s != null);
}
```

Rust

```rust
pub async fn semester_id_list(&self) -> Result<Vec<Id>> {
  let res = self.0.get(SEMESTER_LIST).send().await?.json::<
      Vec<Option<String>>>().await?;
  Ok(res.into_iter().filter_map(|x| x).collect())
}
```

Basic workflow: post your request, get a json or html reply, and extract data from it.

Features

1. Better format checking & error handling.
2. More functionalities: submitting homework, manipulating course discussions, etc.
3. Similar asynchronous API.

Features

1. Better format checking & error handling.
2. More functionalities: submitting homework, manipulating course discussions, etc.
3. Similar asynchronous API.
   ... But there are more to consider, due to the difference in the nature of these two languages.

Contents

The fuse-rs Library

https://github.com/zargony/fuse-rs

The fuse-rs Library

https://github.com/zargony/fuse-rs

User-defined file system operations is abstracted by a Rust trait:

```
pub trait Filesystem {
  /// Initialize filesystem.
  /// Called before any other filesystem method.
  fn init(&mut self, req: &Request) -> Result<(), c_int> {
    Ok(())
  }
  ...
}

pub fn mount<FS: Filesystem, P: AsRef<Path>>(filesystem: FS,
    mountpoint: P, options: &[&OsStr]);
```

The fuse-rs Library

https://github.com/zargony/fuse-rs

User-defined file system operations is abstracted by a Rust trait:

```
pub trait Filesystem {
  /// Initialize filesystem.
  /// Called before any other filesystem method.
  fn init(&mut self, req: &Request) -> Result<(), c_int> {
    Ok(())
  }
  ...
}

pub fn mount<FS: Filesystem, P: AsRef<Path>>(filesystem: FS,
    mountpoint: P, options: &[&OsStr]);
```

> *"fuse-rs does not just provide bindings, it is a rewrite of the original FUSE C library to fully take advantage of Rust's architecture."*

The fuse-rs Library

How does FUSE work:

The fuse-rs Library

How does FUSE work:

1. The kernel driver that registers as a filesystem and forwards operations into a communication channel to a userspace process that handles them.
2. The userspace library that helps the userspace process to establish and run communication with the kernel driver.
3. The userspace implementation that actually processes the filesystem operations.

The fuse-rs Library

How does FUSE work:

1. The kernel driver that registers as a filesystem and forwards operations into a communication channel to a userspace process that handles them.
2. The userspace library that helps the userspace process to establish and run communication with the kernel driver.
3. The userspace implementation that actually processes the filesystem operations.

fuse-rs is not built upon libfuse, instead, it is a replacement of libfuse.

Features

Pros:

Features

Pros:

1. Easy access to private data:

   `&mut self` v.s. `struct fuse_context *fuse_get_context(void);`

   Actually many perfer using global variables...

Features

Pros:

1. Easy access to private data:

   `&mut self` v.s. `struct fuse_context *fuse_get_context(void);`

   Actually many perfer using global variables...

2. No need for path parsing:

   ```
   fn lookup(&mut self, req: &Request, parent: u64, name: &OsStr, reply: ReplyEntry)
   ```
   V.S.
   ```
   int (*open)(const char *, struct fuse_file_info *)
   ```

Features

Pros:

1. Easy access to private data:

   `&mut self` v.s. `struct fuse_context *fuse_get_context(void);`

   Actually many perfer using global variables...

2. No need for path parsing:

   ```
   fn lookup(&mut self, req: &Request, parent: u64, name: &OsStr, reply: ReplyEntry)
   ```
   V.S.
   ```
   int (*open)(const char *, struct fuse_file_info *)
   ```

Cons:

Features

Pros:

1. Easy access to private data:
   `&mut self` v.s. `struct fuse_context *fuse_get_context(void);`
   Actually many perfer using global variables...
2. No need for path parsing:

```
fn lookup(&mut self, req: &Request, parent: u64, name: &OsStr, reply: ReplyEntry)
```

v.s.

```
int (*open)(const char *, struct fuse_file_info *)
```

Cons:

1. Lack of documentation:
   "The readdir implementation ignores the offset parameter, and passes zero to the filler function's offset. The filler function will not return '1' (unless an error happens), so the whole directory is read in a single readdir operation."

Features

Pros:

1. Easy access to private data:
   `&mut self` v.s. `struct fuse_context *fuse_get_context(void);`
   Actually many perfer using global variables...
2. No need for path parsing:

```
fn lookup(&mut self, req: &Request, parent: u64, name: &OsStr, reply: ReplyEntry)
```

v.s.

```
int (*open)(const char *, struct fuse_file_info *)
```

Cons:

1. Lack of documentation:
   "The readdir implementation ignores the offset parameter, and passes zero to the filler function's offset. The filler function will not return '1' (unless an error happens), so the whole directory is read in a single readdir operation."
2. Other small bugs.

Contents

Run Asynchronous Code in Synchronous Context

Usually we use procedural macro `#[tokio::main]` to mark the entry of an asynchronous application, e.g., a web server. But the essence of FUSE has limit our functions to be synchronous.

Run Asynchronous Code in Synchronous Context

Usually we use procedural macro `#[tokio::main]` to mark the entry of an asynchronous application, e.g., a web server. But the essence of FUSE has limit our functions to be synchronous.

Manually submit an asynchronous task to a runtime and wait for the result:

```
self.runtime.block_on(client.discussion_replies(course, discussion, board))
```

Run Asynchronous Code in Synchronous Context

Usually we use procedural macro `#[tokio::main]` to mark the entry of an asynchronous application, e.g., a web server. But the essence of FUSE has limit our functions to be synchronous.

Manually submit an asynchronous task to a runtime and wait for the result:

```
self.runtime.block_on(client.discussion_replies(course, discussion, board))
```

But we can still benefit form asynchronous code: both inside and outside the web learning API we can simultaneously perform multiple requests.

Input Password

In my design, you input your password when you make a directory with its name as the student id in the mounted directory.

Input Password

In my design, you input your password when you make a directory with its name as the student id in the mounted directory.

The thing is, how can you input your password in the `mkdir` process, instead of the FUSE process?

Input Password

In my design, you input your password when you make a directory with its name as the student id in the mounted directory.

The thing is, how can you input your password in the `mkdir` process, instead of the FUSE process?

Use procfs!

https://man7.org/linux/man-pages/man5/proc.5.html

> *"/proc/[pid]/fd/: This is a subdirectory containing one entry for each file which the process has open, named by its file descriptor, and which is a symbolic link to the actual file. Thus, 0 is standard input, 1 standard output, 2 standard error, and so on."*

Upload File

The problem is that you may have to perform a filesystem operation inside your user-defined filesystem operation...

Upload File

The problem is that you may have to perform a filesystem operation inside your user-defined filesystem operation...

https://linux.die.net/man/3/fuse

> *"Note that single-threaded mode also means that you will not have to worry about reentrancy, though you will have to worry about recursive lookups. In single-threaded mode, FUSE holds a global lock on your filesystem, and will wait for one callback to return before calling another. This can lead to deadlocks, if your script makes any attempt to access files or directories in the filesystem it is providing."*

Upload File

Solution: finish current operation, and perform the filesystem operation in another thread.

```
self.runtime.spawn(async move {
  ...
  let dir = Dir::open(format!("/proc/{}/cwd", pid))?;
  let mut file = dir.open_file(path)?;
  ...
});
```

Upload File

Solution: finish current operation, and perform the filesystem operation in another thread.

```
self.runtime.spawn(async move {
  ...
  let dir = Dir::open(format!("/proc/{}/cwd", pid))?;
  let mut file = dir.open_file(path)?;
  ...
});
```

Disadvantage: it is not possible to know whether this operation will succeed when it is finished, so we cannot report it to the user either.

Contents

Rust is both good at CRUD...

```rust
#[derive(Debug, Deserialize)]
pub struct Course {
  #[serde(rename = "wlkcid")] pub id: Id,
  #[serde(rename = "kcm")] pub name: String,
  #[serde(rename = "ywkcm")] pub english_name: String,
  ...
}

pub async fn course_list(&self, semester: IdRef<'_>) -> Result<Vec<Course>> {
  let mut res = self.0.get(&COURSE_LIST(semester)).send().await?.json::<
     JsonWrapper1<Course>>().await?.resultList;
  try_join_all(res.iter_mut().map(async move |x| {
    x.time_location = self.0.get(&COURSE_TIME_LOCATION(&x.id)).send().await?.
     json().await?;
    OK
  })).await?;
  Ok(res)
}
```

...and system programming.

```
fn get_password(pid: u32) -> io::Result<String> {
  let mut stdout = OpenOptions::new().write(true).open(format!(
    "/proc/{}/fd/1", pid))?;
  stdout.write_all("password: ".as_bytes())?;
  stdout.flush()?;
  let mut stdin = BufReader::new(File::open(format!("/proc/{}/
    fd/0", pid))?);
  let mut password = String::new();
  stdin.read_line(&mut password)?;
  Ok(password.trim().to_owned())
}
```

Thanks!

Q & A