

全局变量 + lazy_static

但是，上面的代码不能通过编译，因为 `Arc::new()` 函数无法用于全局变量的初始化。

因此，我们需要引入第三方库 `lazy_static` 来实现这个事情，比如：

```
use std::sync::{Arc, Mutex};
use lazy_static::lazy_static;

struct Job;

lazy_static! {
    static ref JOB_LIST: Arc<Mutex<Vec<Job>>> =
        Arc::new(Mutex::new(Vec::new()));
}
```

这样就可以通过编译了。

互斥锁

那么，在读取或者修改的时候，需要首先获取互斥锁，然后就可以对内部的 `Vec<Job>` 进行操作了：

```
let mut lock = JOB_LIST.lock().unwrap();  
lock.push(Job);
```

为了保证多线程安全，互斥锁保证同时只能有一个线程获取，保证了数据在多线程场景下同时只有一个地方拥有可变借用。当 `lock` 结束生命周期时，锁会自动释放。编写代码的时候，需要小心出现死锁的情况。

死锁

为了防止死锁，这里介绍几种常见的死锁情况：

① 获取锁顺序

```
let lock_a = A.lock().unwrap();  
let lock_b = B.lock().unwrap();
```

```
let lock_b = B.lock().unwrap();  
let lock_a = A.lock().unwrap();
```

获取锁的顺序颠倒，如果上面两侧代码同时运行，可能会出现左侧代码获取了 A 的互斥锁，右侧代码获取了 B 的互斥锁的情况，此时两侧代码都会阻塞在第二次 lock 上。

死锁

② 恐慌

如果获取了锁，但是在释放之前出现了恐慌，会导致锁进入 `Poisoned` 状态：

```
let lock = LOCK.lock().unwrap();
panic!();
```

之后如果其他线程尝试获取这个锁，都会失败。

建议使用互斥锁的时候，尽量缩小持有锁的时间，并且使用的时候尽量不要用 `unwrap()` 等可能出现恐慌的错误处理方法。

死锁

- ③ 模式匹配时，注意不要在匹配内部再次获取锁：

```
match LOCK.lock().unwrap() {
    XX(yy) => {
        // dead lock
        let lock = LOCK.lock().unwrap();
    }
}

if let XX(yy) = LOCK.lock().unwrap() { } else {
    // dead lock
    let lock = LOCK.lock().unwrap();
}
```

如何避免死锁

保证获取锁的代码区域（临界区，Critical Section）尽量小，并且不会出现恐慌：

```
let mut lock = JOB_LIST.lock().unwrap();  
lock.push(Job);  
drop(lock);
```

```
// later
```

```
let mut lock = JOB_LIST.lock().unwrap();  
lock.pop();  
drop(lock);
```


竞争条件

但是，临界区不一定是越小越好：

```
let mut lock = JOB_LIST.lock().unwrap();  
// find maximum job id  
let max_job_id = ???;  
drop(lock);  
  
// later  
let mut lock = JOB_LIST.lock().unwrap();  
lock.push(Job {  
    id: max_job_id + 1  
});  
drop(lock);
```

可能会导致任务 ID 冲突。

JSON 序列化与反序列化

在两个大作业中都出现了 JSON 的序列化和反序列化。你可以使用 `serde_json` 来实现 JSON 的序列化 (Serialize) 与反序列化 (Deserialize), 实现代码中的值与 JSON 值的双向转换:

```
#[derive(Serialize, Deserialize)]    {
struct Config {                      "random": true,
    random: Option<bool>,            "difficult": false,
    difficult: Option<bool>,        "stats": true,
    stats: Option<bool>,           "day": 5,
    day: Option<usize>,            "seed": 20220123,
    seed: Option<u64>,            // ...
    // ...                          }
}
```

在 OJ 大作业中, 涉及到 JSON 的地方会更多, 因此这里介绍一下 `serde_json` 的进阶用法。

枚举与 JSON 的转换

在 OJ 大作业中，为了表示评测任务的状态，会出现枚举：

```
{ "state": "Queueing" }
```

此时，我们可以用枚举类型来表示 `state`：

```
#[derive(Serialize, Deserialize)]
enum State { Queueing, Running, Finished }
```

```
#[derive(Serialize, Deserialize)]
struct Job {
    // ...
    state: State,
}
```


JSON 与 Rust 关键字冲突

还有一种情况，JSON 的键正好对应了 Rust 中的关键字：

```
{ "type": "standard" }
```

```
#[derive(Serialize, Deserialize)]
```

```
struct Problem {  
    // expected identifier, found keyword `type`  
    type: ProblemType,  
}
```

此时有两种方法可以解决：

- ① 把 `type` 改为 `r#type`，强制定义一个名为 `type` 的成员
- ② 把 `type` 改为 `ty`，然后加上标注 `#[serde(rename = "type")]`

如何理解 JSON 比较结果

在 OJ 大作业中，自动测试程序会把 OJ 响应中的 JSON 与预期结果进行比对。于是，你需要学会从错误信息中判断错误出在了什么地方：

例如：

```
json atoms at path ".result" are not equal:
  expected:
    "Wrong Answer"
  actual:
    "Time Limit Exceeded"
```

如何理解 JSON 比较结果

错误信息：

```
json atoms at path ".result" are not equal:
```

```
  expected:
```

```
    "Wrong Answer"
```

```
  actual:
```

```
    "Time Limit Exceeded"
```

表示的是：

实际：

```
{ "result": "Time Limit Exceeded" }
```

期望：

```
{ "result": "Wrong Answer" }
```


子进程

在 OJ 系统中，编译可执行文件以及评测的时候，都需要运行程序，并且按照需求传递命令行参数，或进行输入输出的重定向。Rust 标准库提供了 `std::process::Command` 来运行程序。

首先最简单的用法是，直接运行一个程序，等待其运行结束，并获得它的结束状态：

```
// Taken from rust docs
use std::process::Command;
let status = Command::new("/bin/cat")
    .arg("file.txt")
    .status()?;

println!("process finished with: {status}");
```

上面的代码运行了 `/bin/cat file.txt` 命令，等待其运行完成并获取它的结束状态。

