Introduction to Rust Concurrency

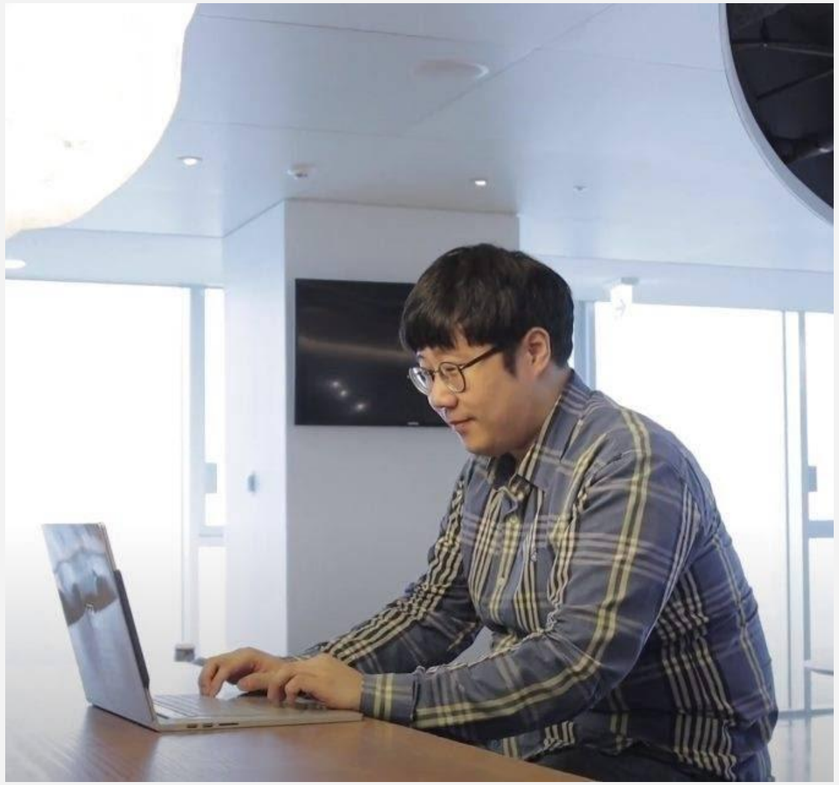
옥찬호 (Chris Ohk)
utilforever@gmail.com

발표자 소개

utilForever@gmail.com

utilForever

- 옥찬호 (Chris Ohk)
  - (현) 42dot Embedded Software Engineer
  - (전) EJN Tech Lead
  - (전) Momenti Engine Engineer
  - (전) Nexon Korea Game Programmer
  - Microsoft Developer Technologies MVP
  - C++ Korea Founder & Administrator
  - Reinforcement Learning KR Administrator
  - IT 전문서 집필 및 번역 다수
  - 대학교 Rust 특강 및 강의 다수



들어가며

- 미리 죄송합니다. (시스템 프로그래밍 수업을 Rust로 한다고 생각하세요.)
- Rust의 기본 기능들을 모르면 코드를 이해하는데 어려울 수 있습니다.
- Rust 동시성은 이렇게 사용하는구나 정도로 보시면 좋습니다.
- 발표 자료와 예제 코드는 다음 저장소에 올릴 예정입니다.

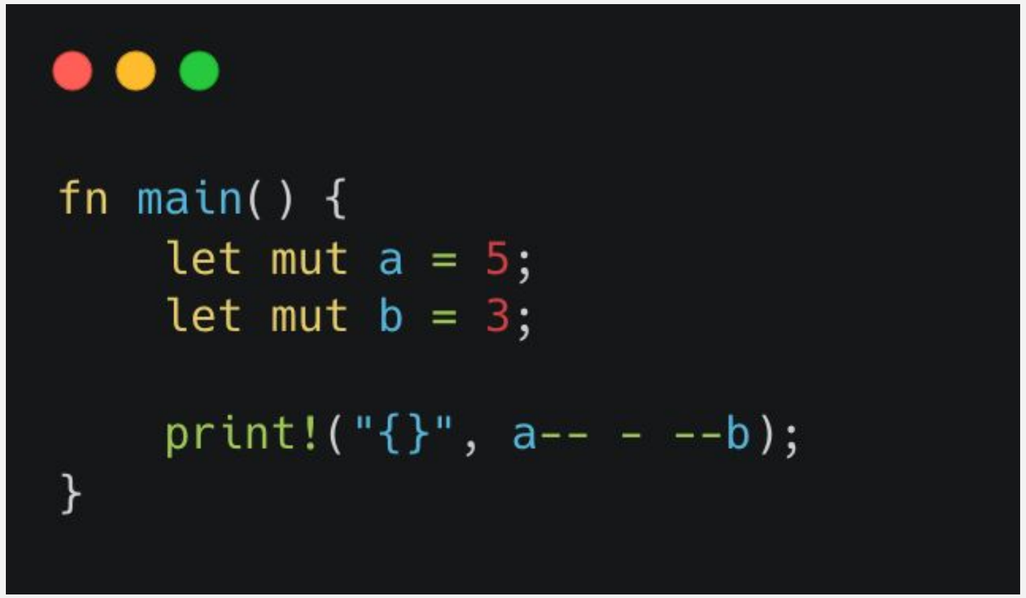
  https://github.com/utilForever/2025-DEVCON-Rust-Concurrency

K DEVCON

Table of Contents

Rust Quiz

- 다음 코드의 출력 결과는? 그리고 이렇게 출력되는 이유는?



```rust
fn main() {
    let mut a = 5;
    let mut b = 3;

    print!("{}", a-- - --b);
}
```

Rust에서 스레드를 사용하는 방법 (1) : Fork Join

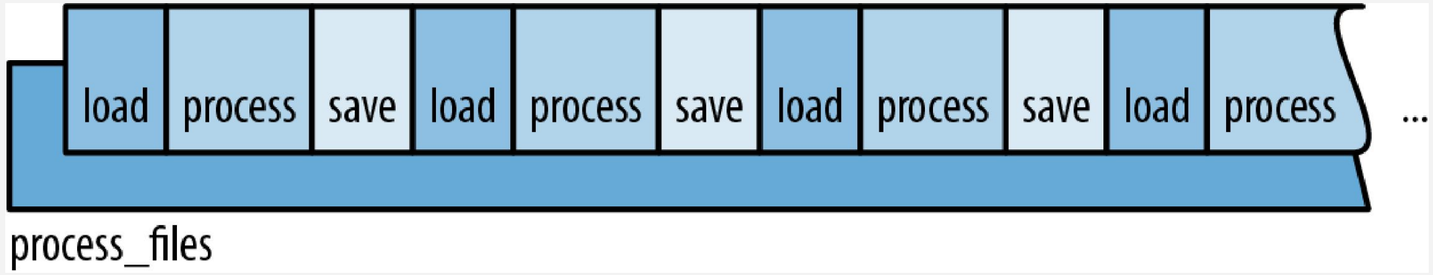
싱글 스레드에서의 프로그램 실행

- 다음과 같은 코드가 있다고 하자.

```rust
fn process_files(filenames: Vec<String>) -> io::Result<()> {
    for document in filenames {
        let text = load(&document)?;
        let results = process(text);

        save(&document, results)?;
    }

    Ok(())
}
```

싱글 스레드에서의 프로그램 실행

- 이 프로그램은 다음과 같이 실행된다.

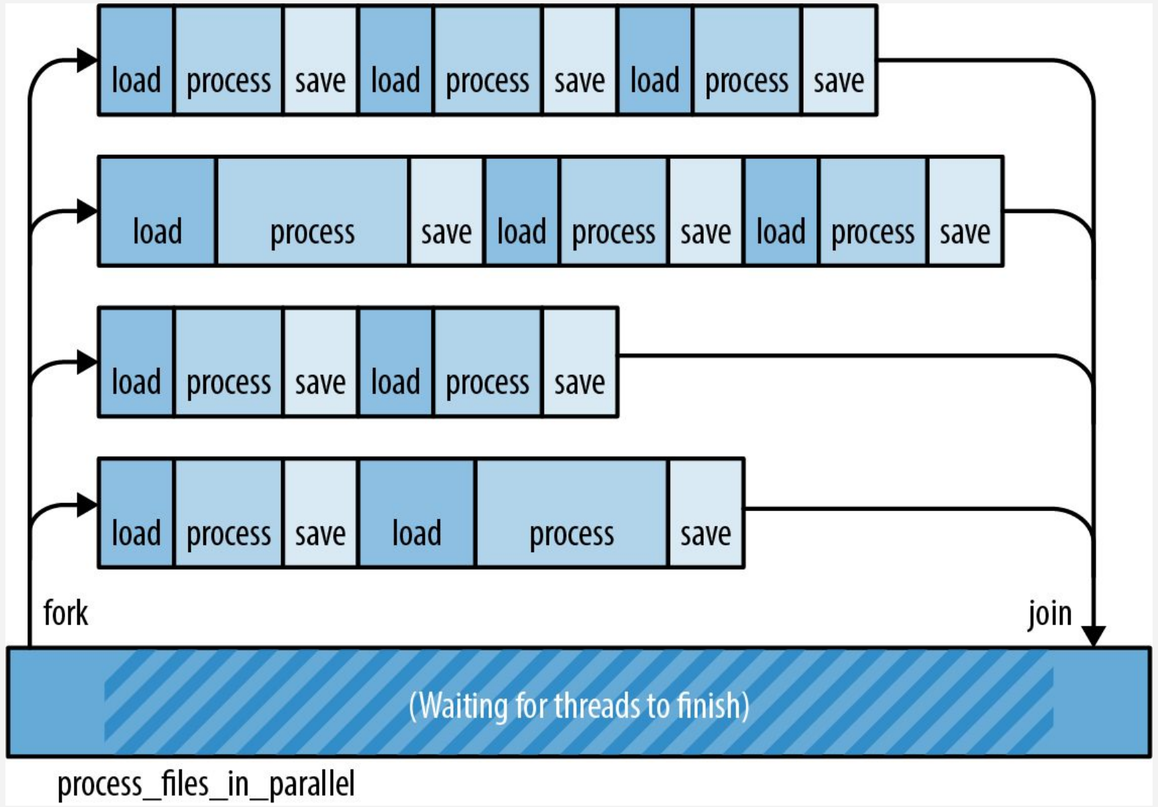


싱글 스레드에서의 프로그램 실행

- 하지만 처리해야 할 파일이 수천 개라면 어떻게 해야 할까?

싱글 스레드에서의 프로그램 실행

- 각 파일은 서로 독립적이므로 나눠서 처리하면 작업 속도를 쉽게 높일 수 있다.



Fork-Join

- 우리는 이 패턴을 **Fork-Join 병렬 처리**라고 한다.
  - **Fork(포크)**는 새 스레드를 시작하는 걸 의미한다.
  - **Join(조인)**은 스레드가 끝나길 기다리는 걸 의미한다.

Fork-Join의 장점

- 구현하기 쉽다.
- 병목 현상이 없다.
- 성능을 예측하기 쉽다.
- 프로그램의 정확성을 추론하기 쉽다.

Fork-Join 써보기

- `std::thread::spawn` 함수를 사용해 새 스레드를 시작한다.

```rust
use std::thread;

thread::spawn(|| {
    println!("Hello from a child thread");
});
```

```rust
use std::{io, thread};

fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
    // Split works into some chunks
    const NUM_THREADS: usize = 8;
    let worklists = split_vec_into_chunks(filename, NUM_THREADS);

    // Fork: Create threads to process the chunks
    let mut thread_handles = vec![];

    for worklist in worklists {
        thread_handles.push(thread::spawn(move || process_files(worklist)));
    }

    // Join: Wait for all threads to finish
    for handle in thread_handles {
        handle.join().unwrap()?;
    }

    Ok(())
}
```

```rust
use std::{io, thread};

fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
    // Split works into some chunks
    const NUM_THREADS: usize = 8;
    let worklists = split_vec_into_chunks(filename, NUM_THREADS);

    // Fork: Create threads to process the chunks
    let mut thread_handles = vec![];

    for worklist in worklists {
        thread_handles.push(thread::spawn(move || process_files(worklist)));
    }

    // Join: Wait for all threads to finish
    for handle in thread_handles {
        handle.join().unwrap()?;
    }

    Ok(())
}
```

`join()` 메소드가 하는 일

- `std::thread::result`를 반환하며, **자식 스레드가 패닉에 빠졌으면** 오류로 간주한다.
  - C++의 기본 동작은 프로세스를 중단하는 것이다.

    Java와 C#의 기본 동작은 자식 스레드의 예외를 터미널에 덤프한 다음 그냥 잊는 것이다.
  - Rust의 패닉은 안전하며, 스레드별로 발생한다.

    스레드 간의 경계는 패닉을 위한 방화벽 역할을 하므로,

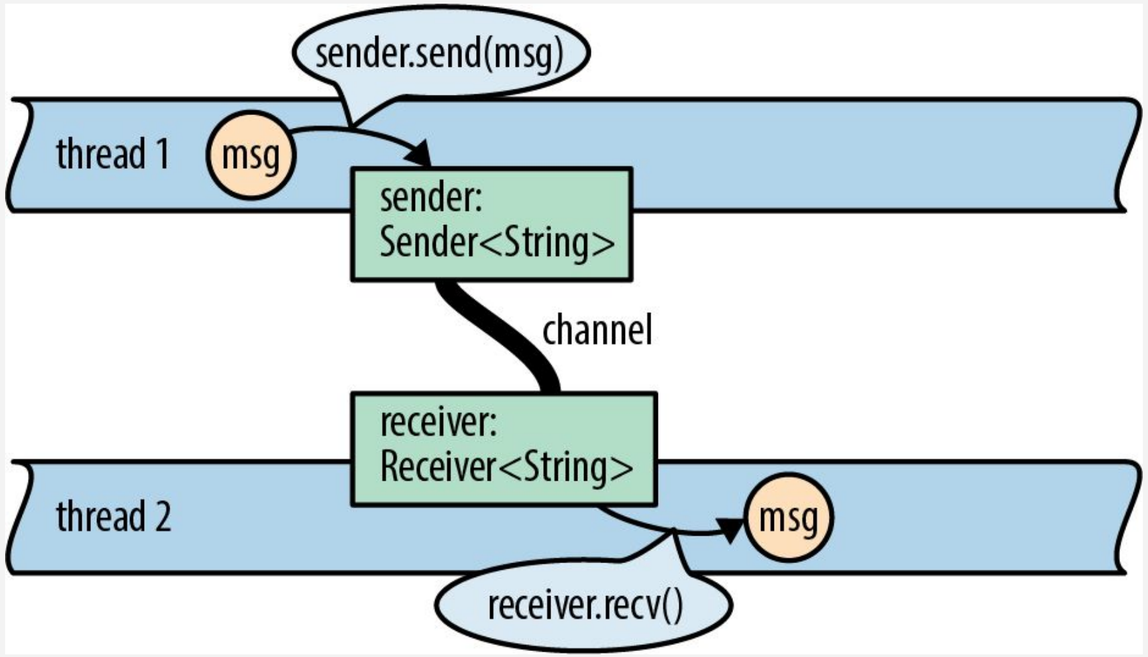
    패닉은 한 스레드에서 그 스레드가 의존하는 다른 스레드로 무작정 퍼지지 않는다.
  - 자식 스레드의 패닉을 부모 스레드에게 전파할 것인가 여부를 직접 선택할 수 있다.

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handles: Vec<_> = (0..2)
        .map(|i| {
            thread::spawn(move || {
                if i == 1 {
                    panic!("Thread {i} panicked!");
                }

                println!("Thread {i} is running");
                thread::sleep(Duration::from_secs(1));
            })
        })
        .collect();

    for handle in handles {
        match handle.join() {
            Ok(_) => println!("Thread completed successfully"),
            Err(e) => {
                println!("Thread failed with error: {:?}", e);
            }
        }
    }

    println!("Main thread continues execution.");
}
```

Rust에서 스레드를 사용하는 방법 (2) : Channel

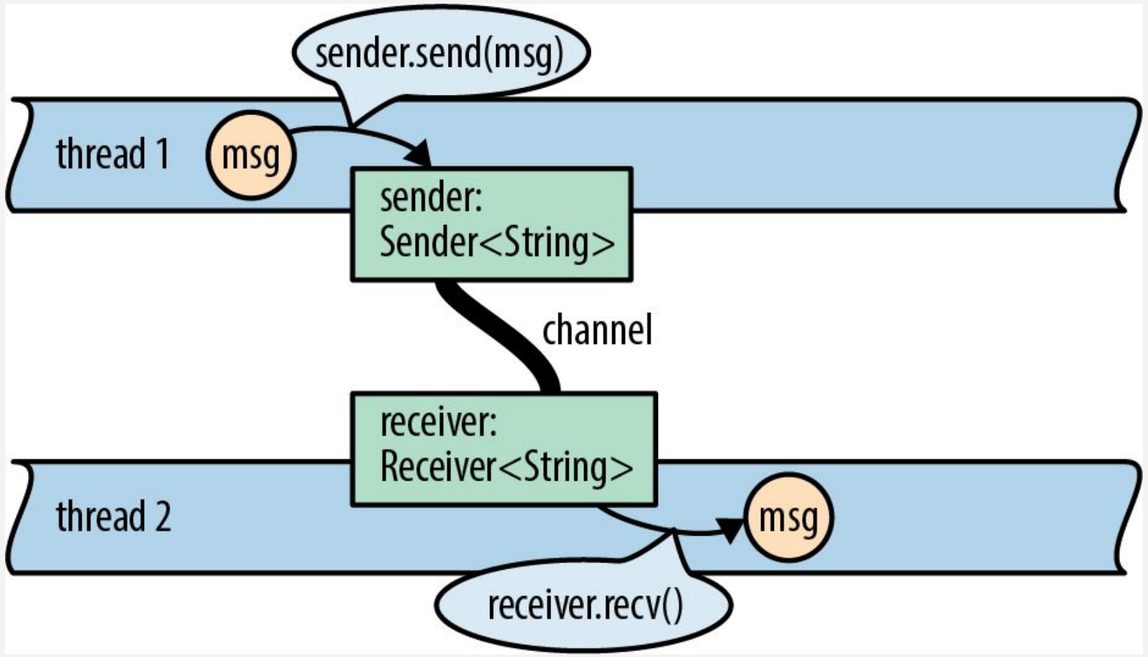
채널

- 한 스레드에서 다른 스레드로 값을 보내기 위한 단방향 도관이다.
  - 스레드 세이프한 큐라고 봐도 무방하다.



채널

- 한쪽 끝은 데이터를 보낼 때 쓰고, 다른쪽 끝은 데이터를 받을 때 쓴다.
  - Unix의 파이프(Pipe)와 비슷하다.



채널

- `sender.send(item)`은 값 하나를 채널에 넣고, `receiver.recv()`는 이 값을 뺀다.
  - 소유권은 보내는 스레드에서 받는 스레드로 넘어간다.
  - 채널이 비었으면 받는 스레드는 누군가 값을 보낼 때까지 블록된다.

값 보내기

- 파일을 읽는 스레드를 생성하는 코드

```rust
use std::sync::mpsc;
use std::{fs, thread};

let (sender, receiver) = mpsc::channel();

let handle = thread::spawn(move || {
    for filename in documents {
        let text = fs::read_to_string(filename)?;

        if sender.send(text).is_err() {
            break;
        }
    }

    Ok(())
});
```

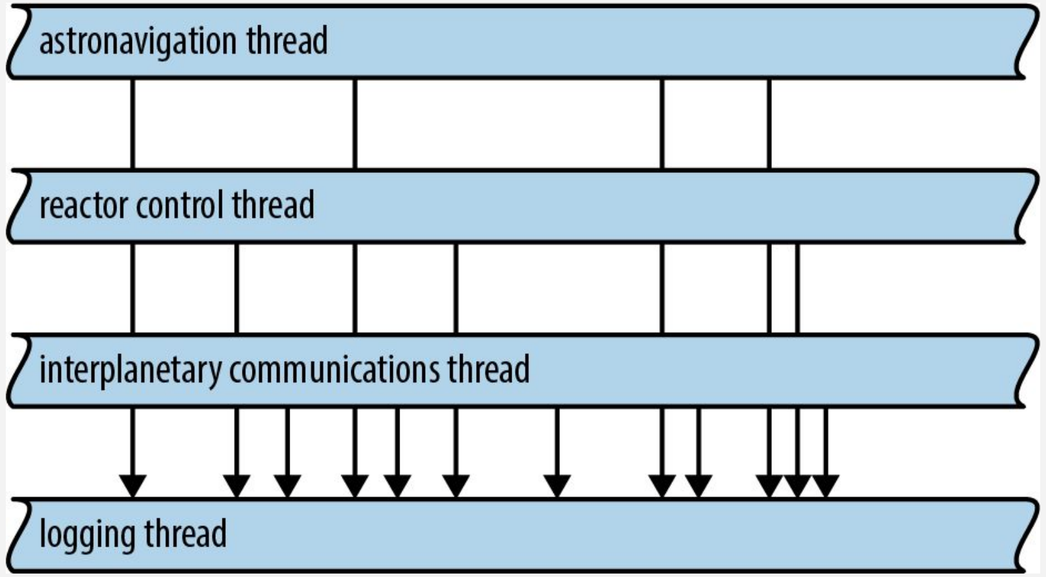
값 받기

- 값을 보내는 반복문을 실행하는 스레드가 준비되었으니,
  이제 `receiver.recv()`를 호출하는 반복문을 실행하면 된다.

```rust
// First
while let Ok(text) = receiver.recv() {
    do_something_with(text);
}

// Second
for text in receiver {
    do_something_with(text);
}
```

채널의 기능

- `std::sync::mpsc`에서 `mpsc`는 무엇일까?
  멀티 프로세서, 싱글 컨슈머(Multi-Producer, Single-Consumer)의 약자로 Rust의 채널이 제공하는 통신의 종류를 한마디로 설명해 준다.



채널의 기능

- Sender<T>는 Clone 트레잇을 구현하고 있다.
  - 보내는 쪽이 여럿인 채널을 얻으려면 일반 채널을 하나 만들고
    보내는 쪽을 원하는 수만큼 복제하면 된다.
  - 각 Sender 값은 다른 스레드로 옮길 수 있다.
- Receiver<T>는 복제할 수 없으므로 같은 채널에서 값을 받는
  여러 스레드가 필요한 경우에는 Mutex가 필요하다.

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let messages = vec!["Hello", "from", "the", "Rust", "channel!"];

        for msg in messages {
            tx.send(msg).unwrap();
            println!("[Producer] Sent: {msg}");
            thread::sleep(Duration::from_millis(500));
        }
    });

    for received in rx {
        println!("[Consumer] Received: {received}");
    }

    println!("All messages received!");
}
```

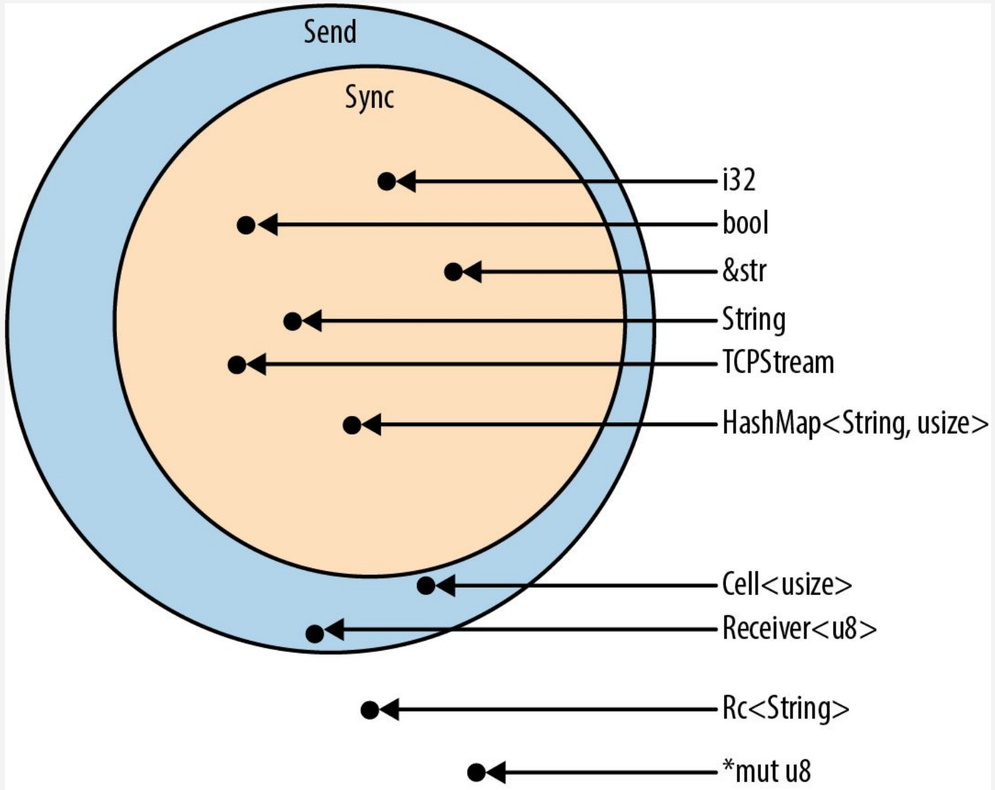
Rust는 어떻게 스레드를 안전하게 만드는가

Send와 Sync

- Rust에서 스레드 안전성과 관련된 이야기는 두 가지 기본 제공 트레잇인 `std::marker::Send`와 `std::marker::Sync`의 여하에 달려 있다.
  - `Send`를 구현하고 있는 타입은 값 전달을 써서 다른 스레드에 넘겨도 안전하다.
    이들은 스레드 간에 이동될 수 있다.
  - `Sync`를 구현하고 있는 타입은 `mut`가 아닌 레퍼런스 전달을 써서 다른 스레드에 넘겨도 안전하다.
    이들은 스레드 간에 공유될 수 있다.
- 안전하다는 건 데이터 경합과 정의되지 않은 동작이 없다는 뜻이다.

Send와 Sync

- 대부분의 타입은 Send면서 Sync다.



예외 사항

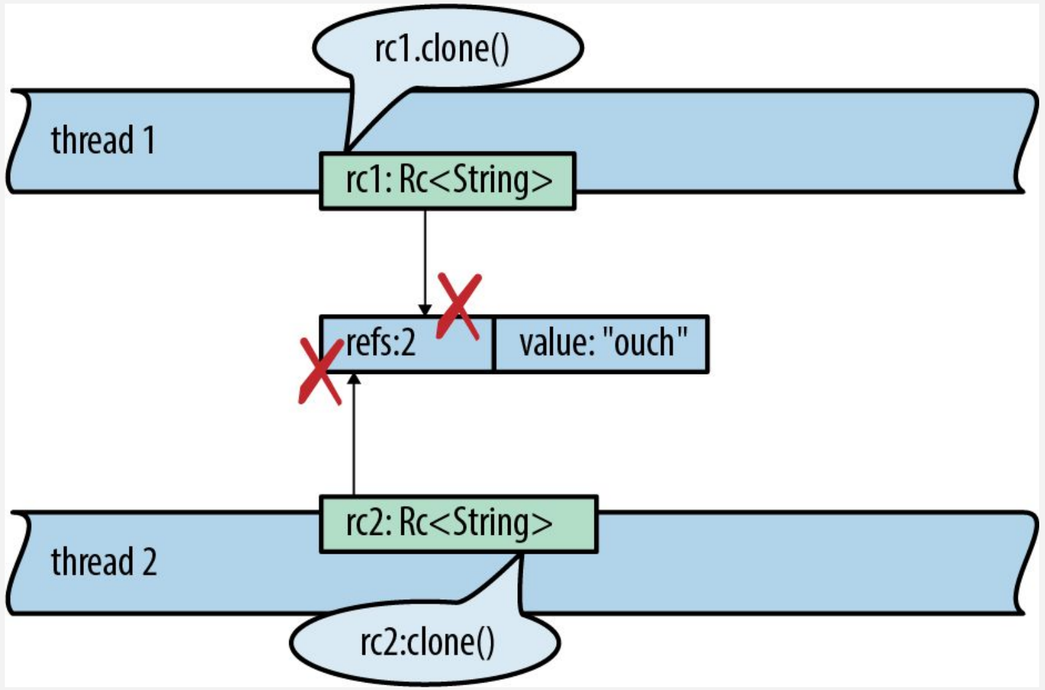
- Send지만 Sync가 아닌 타입도 있는데, 보통 의도적인 결정일 때가 많다.
  `mpsc::Receiver`는 이런 특성을 이용해서 `mpsc` 채널의 받는 쪽을
  한 번에 한 스레드에서만 쓸 수 있게 보장한다.
- 소수지만 Send도 아니고 Sync도 아닌 타입은
  대부분 스레드 세이프하지 않은 방식으로 가변성을 이용한다.

예외 사항 – Rc<T>

- Rc<String>이 Sync여서 Rc 하나를 공유된 레퍼런스를 통해서 스레드 간에 공유할 수 있다면 무슨 일이 벌어질까?

예외 사항 – Rc<T>

- Rc<String>이 Sync여서 Rc 하나를 공유된 레퍼런스를 통해서 스레드 간에 공유할 수 있다면 무슨 일이 벌어질까?



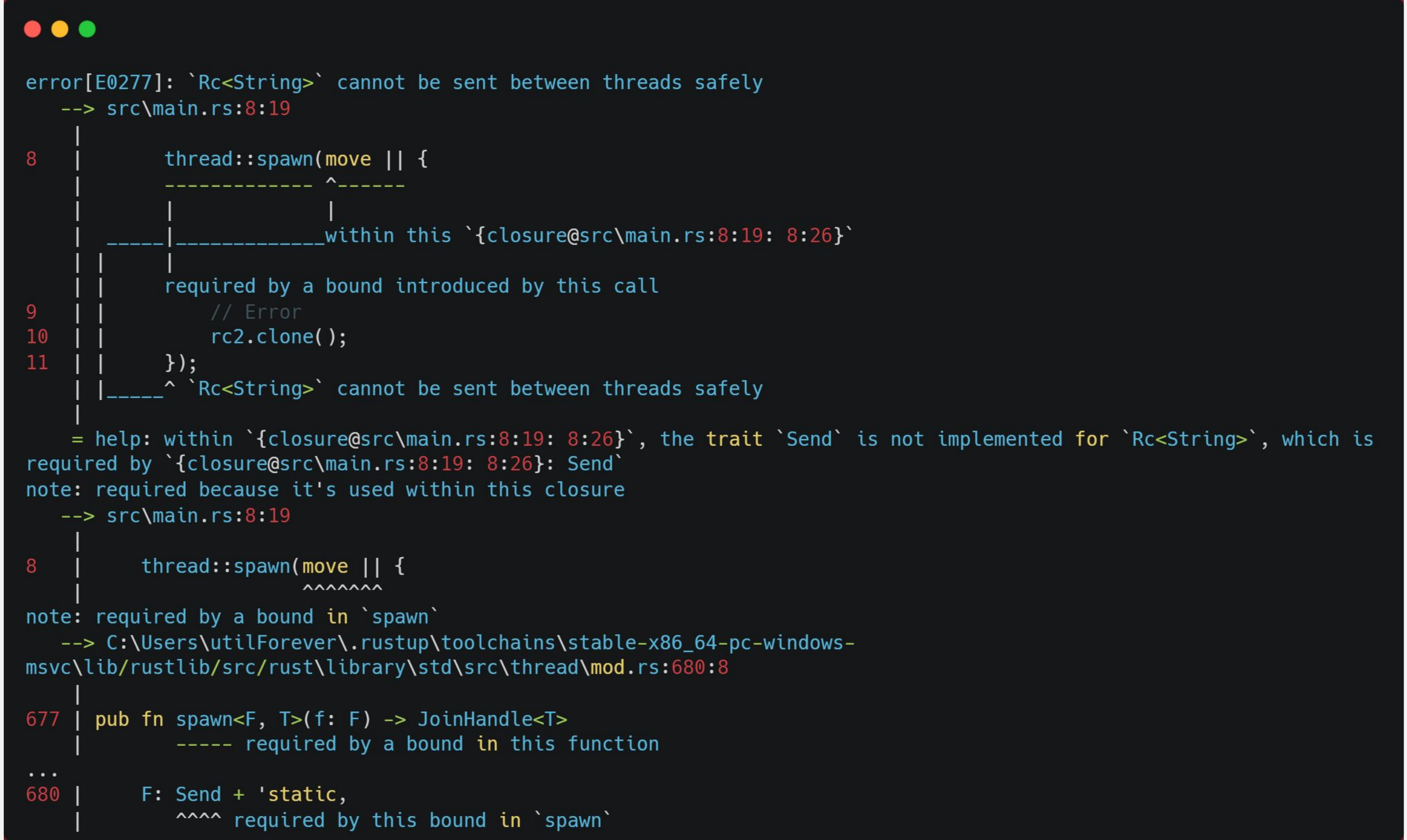
예외 사항 – Rc<T>

- Rc<String>이 Sync여서 Rc 하나를 공유된 레퍼런스를 통해서 스레드 간에 공유할 수 있다면 무슨 일이 벌어질까?
  - 두 스레드가 동시에 Rc를 복제하려고 하면 두 스레드가 공유된 레퍼런스 카운트를 증가시키기 때문에 데이터 경합이 생긴다.
  - 이렇게 되면 레퍼런스 카운트가 부정확해져서 해제 후 사용이나 중복 해제 등 정의되지 않은 동작을 하게 된다.

예외 사항 – Rc<T>

- 따라서 다음 코드는 데이터 경합을 유발해 오류가 발생한다.

```
use std::rc::Rc;
use std::thread;

fn main() {
    let rc1 = Rc::new("ouch".to_string());
    let rc2 = rc1.clone();

    thread::spawn(move || {
        // Error
        rc2.clone();
    });

    rc1.clone();
}
```



```
error[E0277]: `Rc<String>` cannot be sent between threads safely
   --> src\main.rs:8:19
    |
8   |       thread::spawn(move || {
    |       ------------ ^------
    |       |            |
    |  _____|____________within this `{closure@src\main.rs:8:19: 8:26}`
    | |     |
    | |     required by a bound introduced by this call
9   | |         // Error
10  | |         rc2.clone();
11  | |     });
    | |_____^ `Rc<String>` cannot be sent between threads safely
    |
    = help: within `{closure@src\main.rs:8:19: 8:26}`, the trait `Send` is not implemented for `Rc<String>`, which is
required by `{closure@src\main.rs:8:19: 8:26}: Send`
note: required because it's used within this closure
   --> src\main.rs:8:19
    |
8   |     thread::spawn(move || {
    |                   ^^^^^^^
note: required by a bound in `spawn`
   --> C:\Users\utilForever\.rustup\toolchains\stable-x86_64-pc-windows-
msvc\lib/rustlib/src/rust\library\std\src\thread\mod.rs:680:8
    |
677 | pub fn spawn<F, T>(f: F) -> JoinHandle<T>
    |        ----- required by a bound in this function
...
680 |     F: Send + 'static,
    |        ^^^^ required by this bound in `spawn`
```

Rust에서 스레드를 사용하는 방법 (3) : Mutex

뮤텍스란?

- **뮤텍스(Mutex)** 또는 **락(Lock)**은 특정 데이터를 여러 스레드가 필요로 할 때 강제로 번갈아가며 접근하도록 만들기 위해서 쓴다.

뮤텍스가 유용한 이유

- 뮤텍스는 **데이터 경합(Data Race)**,
  즉 바쁘게 돌아가는 여러 스레드가 동시에 같은 메모리를 읽고 쓰는 상황을 방지한다.
  - C++과 Go에서는 데이터 경합이 정의되지 않은 동작이다.
  - Java와 C#에서는 크래시가 발생하지 않는다고 약속하지만,
    데이터 경합의 결과가 무의미한 건 마찬가지다.
- 설령 데이터 경합이 없다고 하더라도,
  또 읽고 쓰는 작업이 전부 프로그램 순서에 따라 차례로 진행되더라도,
  뮤텍스가 없으면 다른 스레드의 동작이 임의의 방식으로 뒤섞일 가능성이 있다.

뮤텍스가 유용한 이유

- 뮤텍스는 **불변성(Invariant)**, 즉 보호되는 데이터가 만족해야 하는 규칙이 있는 프로그래밍을 지원한다. 이 규칙은 데이터를 생성할 때 부여해서 항상 **임계 영역(Critical Section)**으로 유지하고 관리한다.

C++ 뮤텍스의 문제점

- C++에서는 대부분의 언어와 마찬가지로 데이터와 락이 별개의 객체다.
  그래서 “모든 스레드는 데이터를 건드리기 전에 반드시 뮤텍스를 획득하세요.”라고
  주석을 남겨두는 것 외에는 제대로 설명할 방법이 없다.
- 그러나 아무리 주석을 잘 남겨 둬도 컴파일러는 안전한 접근을 강제할 수 없다.
  실수로 뮤텍스를 획득하지 않으면 정의되지 않은 동작이 발생한다.
  이 경우 재현하고 고치기가 극도로 어려운 버그로 남게 된다.

Mutex<T>

- 이제 Rust의 뮤텍스를 살펴 보자.

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });

        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

Rust 뮤텍스

- C++과 달리 보호할 데이터가 Mutex 안에 저장된다.
  - Mutex를 생성하는 건 Box나 Arc를 생성하는 것과 비슷해 보이지만,
    Box와 Arc는 힙 할당을 의미하는 반면 Mutex는 오로지 락에 관한 것이다.
  - Mutex를 힙에 할당하고 싶다면 그렇게 요구하면 되는데,
    전체 애플리케이션에 대해서는 Arc::new를 쓰고 보호할 데이터에 대해서만 Mutex::new를 쓴다.
  - 이들 타입은 같이 쓰일 때가 많은데 Arc는 스레드 간에 뭔가를 공유할 때 유용하고,
    Mutex는 스레드 간에 공유된 변경할 수 있는 데이터가 있을 때 유용하다.

오염된 뮤텍스

- `Mutex::lock()`은 `JoinHandle::join()`과 같은 이유로, 즉 다른 스레드가 패닉에 빠졌을 때 실패를 매끄럽게 처리하기 위해서 `Result`를 반환한다.
  - `handle.join().unwrap()`이라고 쓰면

    Rust에게 한 스레드의 패닉을 다른 스레드로 전파하라고 말하는 것이다.
  - `mutex.join().unwrap()`도 마찬가지다.

오염된 뮤텍스

- 스레드가 `Mutex`를 획득한 상태에서 패닉에 빠지면
  Rust는 그 `Mutex`를 **오염되었다(Poisoned)**고 표시한다.
  - 이 오염된 `Mutex`에 대고 `lock`을 호출하면 `Result`가 `Err`로 반환된다.
  - 이 뒤에 오는 `.unwrap()` 호출은 Rust에 그런 일이 벌어지면 패닉에 빠지라고 말하는 것이다.
  - 따라서 다른 스레드의 패닉이 이쪽으로 전파된다.

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let data = Arc::new(Mutex::new(0));
    let data_clone = Arc::clone(&data);

    let handle = thread::spawn(move || {
        let mut num = data_clone.lock().unwrap();
        *num += 1;

        panic!("Intentional panic: Mutex is poisoned!");
    });

    let _ = handle.join();

    match data.lock() {
        Ok(guard) => println!("Mutex is normal. value: {}", *guard),
        Err(poisoned) => {
            println!("Mutex is poisoned.");

            let guard = poisoned.into_inner();
            println!("Poisoned value: {}", *guard);
        }
    };
}
```

Mutex의 대체재 : RwLock

읽기/쓰기 락

- 서버 프로그램은 한 번 읽어오면 거의 바뀔 일이 없는 구성 정보를 가지고 있을 때가 많다.
  - 대부분의 스레드는 구성을 조회하는데 그치겠지만,
    구성이 바뀔 가능성이 열려 있으므로 반드시 락을 써서 보호해야 한다.
  - 뮤텍스는 이런 경우에도 통하겠지만 불필요한 병목을 만든다.
    스레드가 구성을 바꾸지 않고 조회하기만 한다면 굳이 줄을 세워 관리할 이유가 없다.
  - 이럴 때 쓰라고 만든 게 바로 읽기/쓰기 락, 즉 `RwLock`이다.

읽기/쓰기 락

- 읽는 메소드는 `RwLock::read()`를 쓴다.

```rust
use std::sync::{Arc, RwLock};
use std::thread;

fn main() {
    let numbers = Arc::new(RwLock::new(vec![10, 20, 30, 40, 50]));
    let mut handles = vec![];

    for i in 0..5 {
        let numbers_clone = Arc::clone(&numbers);
        let handle = thread::spawn(move || {
            let nums = numbers_clone.read().unwrap();
            println!("Thread {i}: numbers = {:?}", *nums);
        });

        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

읽기/쓰기 락

- 쓰는 메소드는 `RwLock::write()`를 쓴다.

```rust
use std::sync::{Arc, RwLock};
use std::thread;

fn main() {
    let counter = Arc::new(RwLock::new(0));
    let mut handles = vec![];

    for i in 0..10 {
        let counter_clone = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter_clone.write().unwrap();
            *num += 1;

            println!("Thread {i}: counter = {}", *num);
        });

        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Final counter value: {}", *counter.read().unwrap());
}
```

특정 조건이 필요하다면 : Condvar

조건 변수

- 스레드는 특정 조건이 참이 될 때까지 기다려야 하는 경우가 많다.
  - 서버가 종료하는 과정에서 메인 스레드는 다른 스레드가 전부 일을 마칠 때까지 기다려야 할 수 있다.
  - 워커 스레드가 할 일이 없을 때는 처리할 데이터가 생길 때까지 기다려야 한다.
  - 분산 합의 프로토콜을 구현하고 있는 스레드는 응답한 피어(Peer) 수가 정족수를 채울 때까지 기다려야 할 수 있다.

조건 변수

- 이럴 때는 **조건 변수(Condition Variable)**를 써서
  프로그램 안에 직접 원하는 조건을 만들 수 있다.
  - Rust에서는 `std::sync::Condvar` 타입이 조건 변수를 구현하고 있다.
  - `Condvar`에는 `.wait()`와 `.notify_all()` 메소드가 있다.
    `.wait()`는 다른 스레드가 `.notify_all()`을 호출할 때까지 블록된다.

조건 변수

- 원하는 조건이 참이 되면 `Condvar::notify_all`(또는 `notify_one`)을 호출해서 대기 중인 스레드를 전부 깨운다.

```
self.has_data_condvar.notify_all();
```

- 조건이 참이 될 때까지 잠든 채로 대기하려면 `Condvar::wait()`을 쓴다.

```
while !guard.has_data() {
    guard = self.has_data_condvar.wait(guard).unwrap();
}
```

```rust
use std::collections::VecDeque;
use std::sync::{Arc, Condvar, Mutex};
use std::thread;
use std::time::Duration;

const BUFFER_SIZE: usize = 5;

struct SharedQueue<T> {
    queue: Mutex<VecDeque<T>>,
    condvar: Condvar,
}

fn main() {
    let shared_queue = Arc::new(SharedQueue {
        queue: Mutex::new(VecDeque::new()),
        condvar: Condvar::new(),
    });

    let producer_queue = Arc::clone(&shared_queue);
    let consumer_queue = Arc::clone(&shared_queue);

    let producer = ...
    let consumer = ...

    producer.join().unwrap();
    consumer.join().unwrap();
}
```

```rust
    let producer = thread::spawn(move || {
        for i in 0..20 {
            let mut queue = producer_queue.queue.lock().unwrap();

            while queue.len() >= BUFFER_SIZE {
                println!("Producer is waiting: The buffer is full.");
                queue = producer_queue.condvar.wait(queue).unwrap();
            }

            queue.push_back(i);
            println!("Producer: produce {i}");

            producer_queue.condvar.notify_all();

            thread::sleep(Duration::from_millis(100));
        }
    });
```

```rust
    let consumer = thread::spawn(move || {
        for _ in 0..20 {
            let mut queue = consumer_queue.queue.lock().unwrap();

            while queue.is_empty() {
                println!("Consumer is waiting: The buffer is empty.");
                queue = consumer_queue.condvar.wait(queue).unwrap();
            }

            let item = queue.pop_front().unwrap();
            println!("Consumer: consume {item}");

            consumer_queue.condvar.notify_all();

            thread::sleep(Duration::from_millis(150));
        }
    });
```

원자적 연산이 필요하다면 : Atomic

원자성

- `std::sync::atomic` 모듈에는 락이 없는 동시성 프로그래밍을 위한 원자적 타입이 포함되어 있다.
  - 일부 추가 기능을 제외하면 기본적으로 C++의 원자성 지원과 동일하다.
  - `AtomicIsize`, `AtomicUsize`
  - `AtomicI8`, `AtomicI16`, `AtomicI32`, `AtomicI64`
  - `AtomicBool`
  - `AtomicPtr<T>`

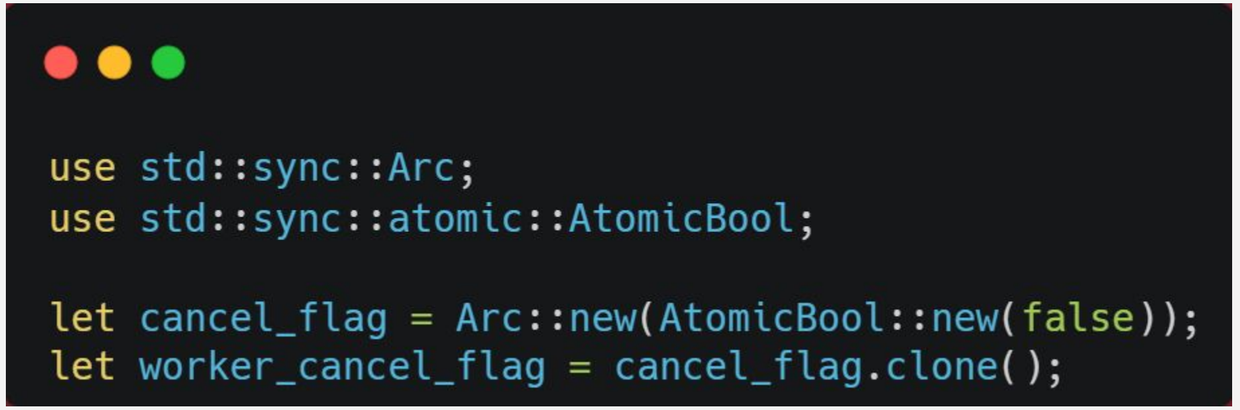
원자성

- 원자적 타입은 평범한 산술과 논리 연산자가 아니라
  원자적 연산(Atomic Operation)을 수행하는 메소드를 사용한다.
  - 각기 다른 로드, 스토어, 익스체인지를 비롯해서 다른 스레드가 같은 메모리 위치를 건드리는 원자적 연산을 수행하더라도 하나의 단위로 안전하게 처리되는 산술 연산이 포함된다.
  - 예를 들어, `atom`이라는 이름의 `AtomicIsize`를 증가시키려면 다음처럼 하면 된다.

```
use std::sync::atomic::{AtomicIsize, Ordering};

let atom = AtomicIsize::new(0);
atom.fetch_add(1, Ordering::SeqCst);
```

원자성

- 원자성의 쉬운 예로 취소가 있다.
  - 예를 들어, 비디오 렌더링과 같이 오래 걸리는 계산을 수행하는 스레드가 있는데
    이를 비동기적으로 취소할 수 있게 만들고 싶다고 하자.
  - 문제는 종료시킬 스레드와 어떤 식으로 통신해야 좋을 지 모른다는 건데,
    이럴 때 공유된 `AtomicBool`을 쓰면 간단히 해결할 수 있다.



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

let cancel_flag = Arc::new(AtomicBool::new(false));
let worker_cancel_flag = cancel_flag.clone();
```

원자성

- 워커 스레드는 다음과 같다.

```rust
use std::sync::atomic::Ordering;
use std::thread;

let worker_handle = thread::spawn(move || {
    for pixel in animation.pixels_mut() {
        // Ray-tracing - this takes a few microseconds
        render(pixel);

        if worker_cancel_flag.load(Ordering::SeqCst) {
            return None;
        }
    }

    Some(animation)
});
```

원자성

- 메인 스레드가 워커 스레드를 취소하기로 결정하면 `AtomicBool`에 `true`를 저장하고 스레드가 종료될 때까지 기다린다.

```rust
// Cancel rendering
cancel_flag.store(true, Ordering::SeqCst);

// Discard the result, which is probably `None`
worker_handle.join().unwrap();
```

정리

- 핵심 개념 요약
  - Fork-Join: 작업 분할과 병렬 처리의 기본 패턴
  - Channel: 안전한 메시지 전달을 통한 스레드 간 통신
  - Mutex & Condvar: 공유 자원 접근과 조건 동기화
  - Atomic: 락 없이 수행 가능한 원자적 연산
- Rust의 동시성 안전성
  - 컴파일 타임 검증: 소유권과 빌림 시스템으로 데이터 경쟁 예방
  - 런타임 안정성: 패닉 발생 시 뮤텍스 오염 처리 등 안전한 오류 처리 메커니즘

Thank You

옥찬호

42dot / Embedded Software Engineer

utilforever@gmail.com / @utilforever