

HSPACE Rust 특강

Rust Basic #14 - 동시성

Chris Ohk

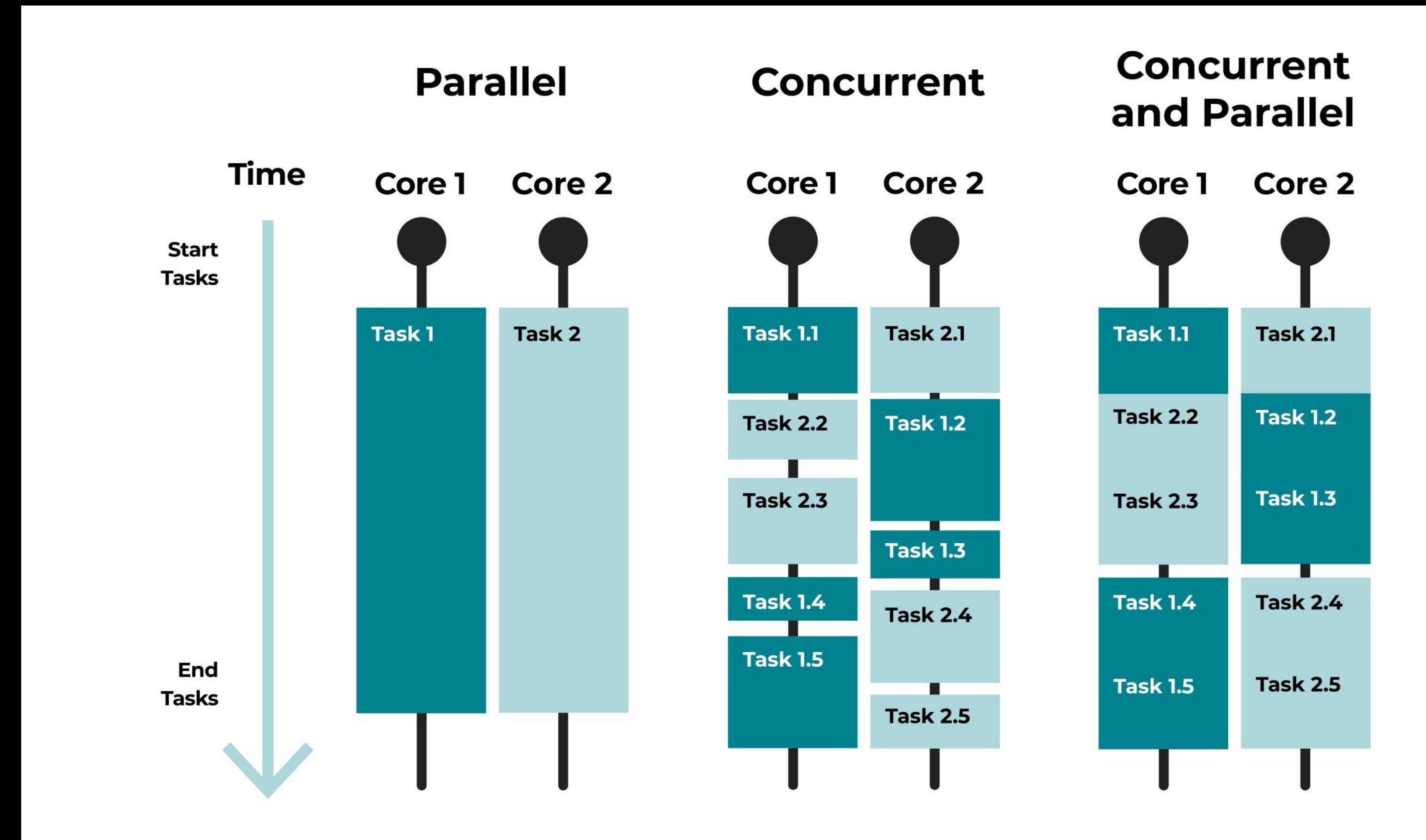
utilForever@gmail.com

- Fork-Join 병렬 처리
 - spawn과 join
 - 스레드 간 오류 처리
 - 스레드 간에 변경할 수 없는 데이터 공유하기
 - Rayon 라이브러리

- 채널
 - 값 보내기
 - 값 받기
 - 파이프라인 실행하기
 - 채널의 기능과 성능
 - 스레드 안전성 : Send와 Sync
 - 거의 모든 이터레이터를 연결해 쓸 수 있는 채널
 - 파이프라인 이외의 다른 채널 사용처

- 변경할 수 있는 공유된 상태
 - 뮤텍스란?
 - `Mutex<T>`
 - `mut`와 `Mutex`
 - 언제 어디서나 뮤텍스만 고집하는 그대에게
 - 교착 상태
 - 오염된 뮤텍스
 - 뮤텍스를 이용한 멀티 컨슈머 채널
 - 읽기/쓰기 락 (`RwLock<T>`)
 - 조건 변수 (`Condvar`)
 - 원자성
 - 전역 변수

- Concurrency vs Parallelism



- Rust에서 스레드를 사용하는 세 가지 방법
 - Fork-Join 병렬처리
 - 채널
 - 변경할 수 있는 공유된 상태

Fork-Join 병렬 처리

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 스레드를 필요로 하는 가장 단순한 사용 사례는 동시에 수행하고 싶은 여러 작업이 서로 완전히 분리되어 있는 경우다.
- 예를 들어, 문서에 있는 방대한 양의 말뭉치를 대상으로 자연어 처리를 수행하고 있다고 하자. 이를 위해서는 다음과 같은 반복문이 필요하다.

```
● ● ●

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

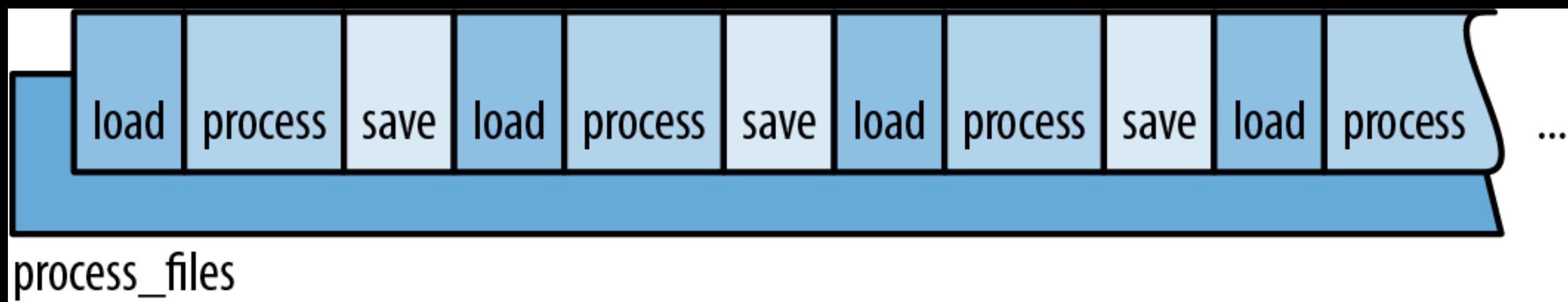
        save(&document, results)?;
    }

    Ok(())
}
```

Fork-Join 병렬 처리

HSPACE Rust 특강
Rust Basic #14 - 동시성

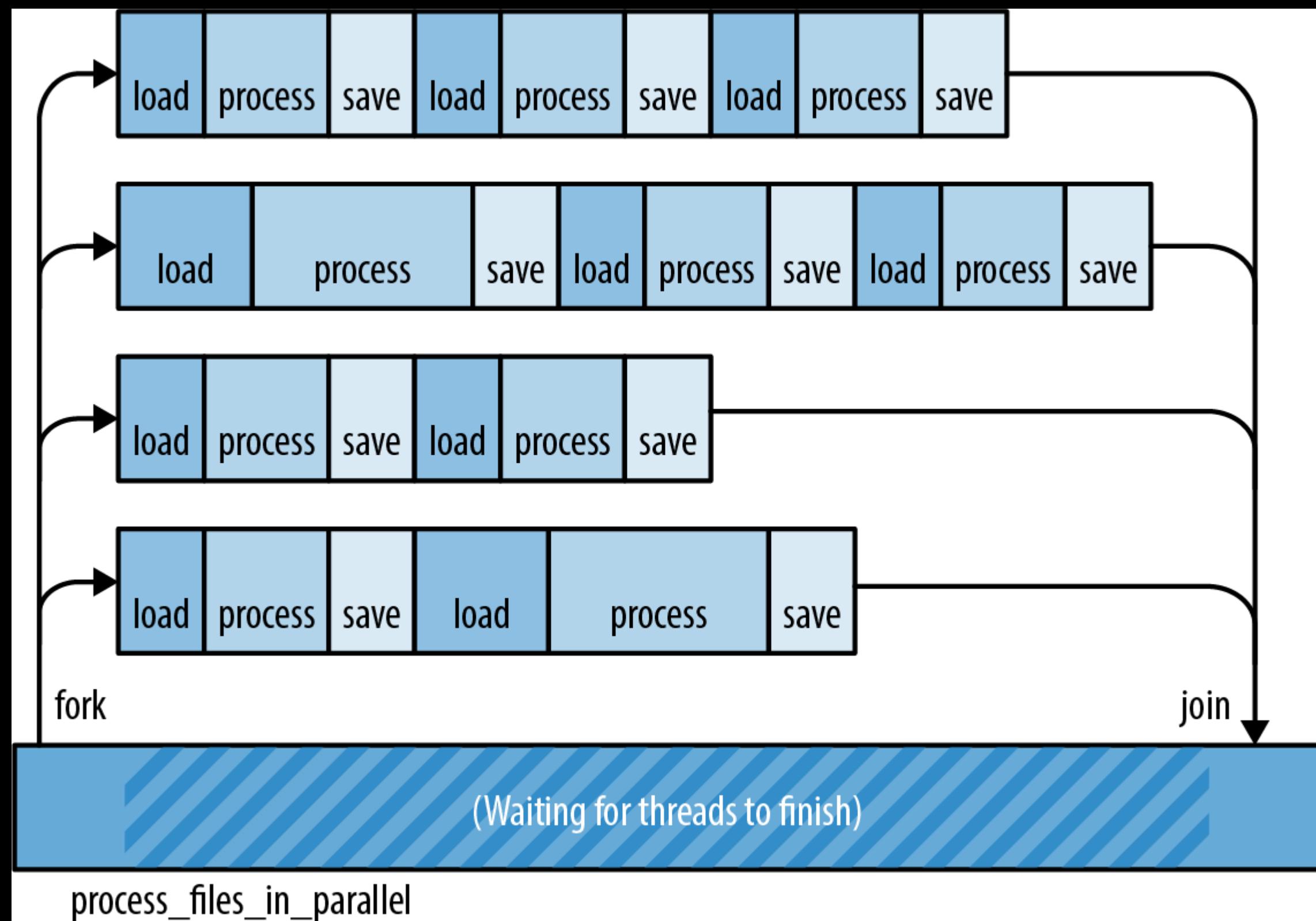
- 이 프로그램은 다음 그림과 같이 실행된다.



Fork-Join 병렬 처리

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 각 문서는 독립적으로 처리되기 때문에, 다음과 같이 말뭉치를 몇 개의 덩어리로 분할해서 각 덩어리를 별도의 스레드가 처리하게 만들면 작업 속도를 쉽게 높일 수 있다.



Fork-Join 병렬 처리

HSPACE Rust 특강
Rust Basic #14 - 동시성

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

Fork-Join 병렬 처리

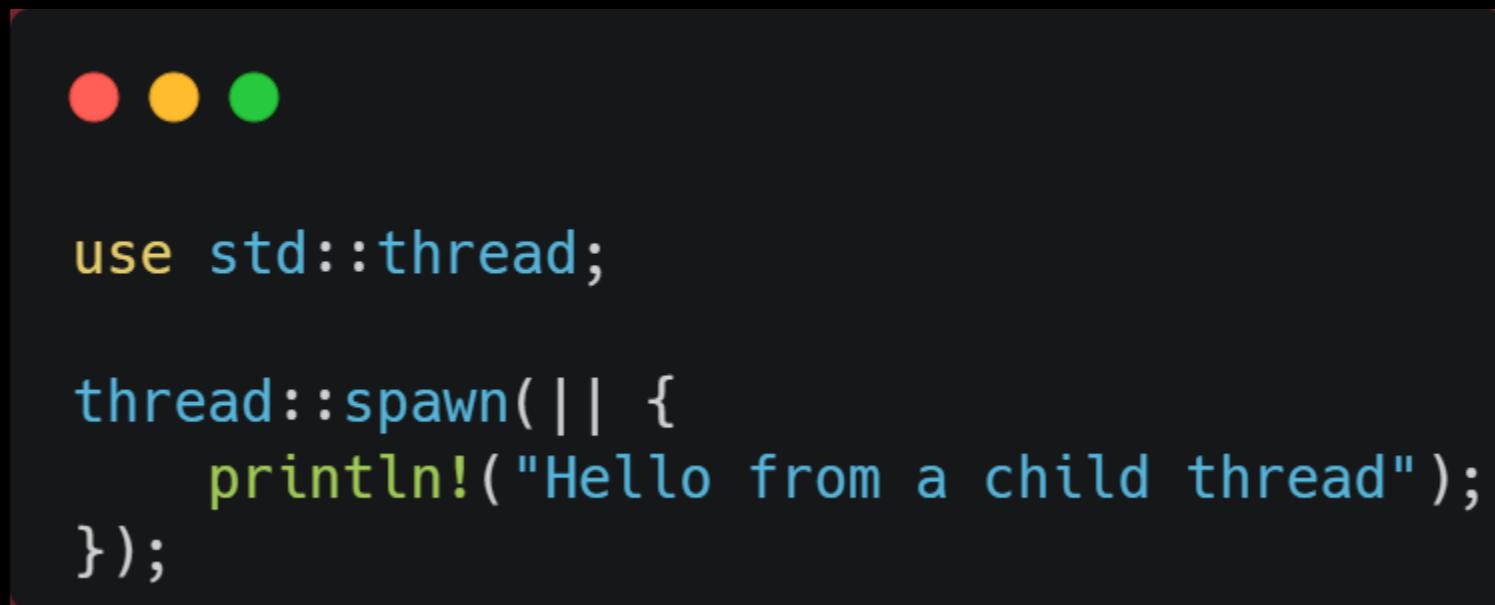
HSPACE Rust 특강
Rust Basic #14 - 동시성

- Fork-Join 병렬 처리가 매력적인 이유는 다음과 같다.
 - 아주 단순하다.
 - Fork-Join은 구현하기 쉬운데다 Rust 덕분에 실수할 여지도 적다.
 - 병목 현상이 없다.
 - 포크-조인에서는 공유된 자원을 잠그는 일이 없다. 한 스레드가 다른 스레드를 기다려야 하는 유일한 시점은 마칠 때 뿐이다. 그동안은 각 스레드가 자유롭게 실행된다. 이 부분은 작업 전환 비용을 낮게 유지하는 데 도움이 된다.
 - 성능을 예측하기 쉽다.
 - 앞서 언급한 작업을 4개의 스레드로 수행하면 최선의 경우에 원래 시간의 1/4만 써서 마칠 수 있다. 그러나 이상적인 경우는 기대하기 힘든데, 그 이유는 작업을 모든 스레드에 고루 분배하지 못할 수도 있기 때문이다.
 - Fork-Join 프로그램에서는 경우에 따라서 스레드를 Join한 이후에 각 스레드가 계산한 결과를 결합하는 시간이 꼭 필요하다. 즉, 작업을 완전히 분리하는 데는 약간의 추가 작업이 필요할 수도 있다는 뜻이다.
 - 프로그램의 정확성을 추론하기 쉽다.
 - Fork-Join 프로그램은 스레드가 실제로 격리되어 있는 한 결정성(Deterministic)을 띤다. 이런 프로그램은 스레드 속도의 변화에 상관없이 항상 같은 결과를 낸다. 경합 상태가 없는 동시성 모델이다.

spawn과 join

HSPACE Rust 특강
Rust Basic #14 - 동시성

- `std::thread::spawn` 함수는 새 스레드를 시작한다.
 - FnOnce 클로저나 함수 하나를 인수로 받는다. Rust는 이 클로저나 함수의 코드를 실행하기 위해서 새 스레드를 시작한다. 새 스레드는 C++, C#, Java에 있는 스레드와 마찬가지로 자체 스택을 가진 실제 운영체제 스레드다.



```
use std::thread;

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

The screenshot shows a terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top. The terminal displays the following Rust code:

```
use std::thread;

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

After running the code, the terminal outputs:

```
Hello from a child thread
```

spawn과 join

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 다음은 spawn을 사용해서 앞에 있는 process_files 함수를 병렬 처리로 구현한 것이다.

```
● ● ●

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(())
}
```

스레드 간 오류 처리

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 앞의 예제에서 자식 스레드를 Join하는 데 썼던 코드는 오류 처리 때문에 보기보다 까다롭다.
- `.join()` 메소드는 두 가지 일을 대신 맡아 처리해 준다.
 - 첫 번째로 `handle.join()`은 `std::thread::Result`를 반환하며, 자식 스레드가 패닉에 빠졌으면 오류로 간주한다. Rust의 스레딩이 C++보다 훨씬 더 견고한 이유가 바로 여기에 있다. C++에서는 배열의 범위 밖에 있는 요소에 접근하는 행위가 정의되지 않은 동작(Undefined Behavior)이라서 결과로부터 나머지 시스템을 보호해낼 재간이 없다. Rust의 패닉은 안전하며, 스레드별로 발생한다. 스레드 간의 경계는 패닉을 위한 방화벽 역할을 하므로, 패닉은 한 스레드에서 그 스레드가 의존하는 다른 스레드로 무작정 퍼지지 않는다. 그게 아니라 한 스레드의 패닉은 `Result`의 값이 `Err`로 다른 스레드에 보고된다. 따라서 프로그램이 전반적으로 쉽게 회복할 수 있다.
 - 그러나 예제에서는 아무런 패닉 처리도 시도하지 않고 있다. 그 대신 직접 이 `Result`에 대고 `.unwrap()`을 써서 결과가 `Err`이 아니라 `Ok`라고 단언한다. 이렇게 하면 자식 스레드가 패닉에 빠질 때 이 단언문이 실패하게 되므로 부모 스레드도 패닉에 빠진다. 자식 스레드의 패닉을 부모 스레드에게 명시적으로 전파하고 있는 것이다.

스레드 간 오류 처리

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 앞의 예제에서 자식 스레드를 Join하는 데 썼던 코드는 오류 처리 때문에 보기보다 까다롭다.
- `.join()` 메소드는 두 가지 일을 대신 맡아 처리해 준다.
 - 두 번째로 `handle.join()`은 자식 스레드의 반환값을 다시 부모 스레드에게 넘긴다. 우리가 `spawn`에 넘긴 클로저의 반환 타입은 `process_files`의 반환 타입인 `io::Result<()>`이다. 이 반환값은 폐기되지 않는다. 자식 스레드가 끝날 때 `process_files`의 반환값은 저장되며, `JoinHandle::join()`이 그 값을 다시 부모 스레드로 옮긴다.
 - 프로그램에서 `handle.join()`이 반환하는 전체 타입은 `std::thread::Result<std::io::Result<()>>`이다. 여기서 `thread::Result`는 `spawn/join` API의 일부이고, `io::Result`는 프로그램의 일부다.
 - 예제 코드에서는 `thread::Result`를 풀어낸 다음 `io::Result`에 대고 ? 연산자를 써서 자식 스레드의 I/O 오류를 부모 스레드에게 명시적으로 전파한다.

스레드 간 오류 처리

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 앞의 예제에서 자식 스레드를 Join하는 데 썼던 코드는 오류 처리 때문에 보기보다 까다롭다.
 - 이 모든 게 다소 복잡해 보일 수도 있다. 그러나 코드 한 줄로 할 수 있는 일이란 걸 염두에 두고 다른 언어와 비교해 보자.
 - Java와 C#의 기본 동작은 자식 스레드의 예외를 터미널에 덤프한 다음 그냥 잊는 것이다.
C++의 기본 동작은 프로세스를 중단하는 것이다.
 - Rust에서는 오류가 예외가 아니라 Result 값이라서 다른 값과 마찬가지로 스레드 간에 전달된다.
그 덕분에 저수준 스레딩 API를 사용할 때는 어떤 식으로는 오류 처리 코드를 꼼꼼히 작성할 수 밖에 없는데,
꼭 작성해야 하는 부분이라는 점을 감안하면 Result를 채택한 디자인은 아주 탁월한 결정이라고 말할 수 있다.

변경할 수 없는 데이터 공유하기

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 분석을 진행할 때 방대한 양의 영어 단어와 구문이 담긴 DB가 필요하다고 하자.

```
// Original version
fn process_files(filenames: Vec<String>)

// Changed version
fn process_files(filenames: Vec<String>, glossary: &GigabyteMap)
```

- 바뀐 함수에서 `glossary`는 덩치가 클 게 뻔하므로 여기서는 레퍼런스로 전달해서 넘긴다.
그렇다면 `process_files_in_parallel`에서 `glossary`을 워커 스레드에게 넘기려면
어디를 어떻게 바꿔야 할까?

변경할 수 없는 데이터 공유하기

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 다음처럼 바꾸면 되는 거 아닐까 싶겠지만, 생각처럼 잘 되지 않는다.

```
● ● ●

fn process_files_in_parallel(filename: Vec<String>, glossary: &GigabyteMap) -> io::Result<()> {
    // ...

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

    // ...
}
```

변경할 수 없는 데이터 공유하기

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 왜 컴파일 오류가 발생할까?
 - Rust가 문제 삼는 부분은 spawn에 넘기고 있는 클로저의 수명으로, 컴파일러가 '도움이 될까 싶어' 제시한 앞의 메시지는 사실 아무런 도움이 되지 않는다.
 - spawn은 독립된 스레드를 띄운다. Rust는 자식 스레드가 얼마나 오래 실행될 지 알 길이 없으므로 최악의 경우를 상성해서, 부모 스레드가 끝나고 부모 스레드의 모든 값이 사라진 뒤에도 자식 스레드가 계속 실행될 수 있다고 가정한다. 자식 스레드가 그렇게 오래 지속된다면 실행 중인 클로저도 당연히 그만큼 오래 지속되어야 한다. 그러나 앞 클로저는 glossary 레퍼런스에 의존하고 있는 데다 레퍼런스란 게 애초부터 영원히 지속될 수 없는 성질을 띠고 있어서 한정된 수명을 갖는다.
 - 결론적으로 Rust는 이 코드를 거부하는 게 맞다.
이 코드대로라면 한 스레드가 I/O 오류를 일으킬 때 process_files_in_parallel이 다른 스레드가 끝나기도 전에 서둘러 떠날 가능성이 있다. 이렇게 되면 자식 스레드는 결국 메인 스레드가 이미 해제한 glossary를 사용하려고 하는 상황에 놓일 수 있다. 그러다 엎친 데 덮친 격으로 메인 스레드가 glossary를 사용하려고 하는 상황에 봉착하면 경합 상태에 빠져서 정의되지 않은 동작을 하게 되는 어처구니없는 일이 벌어진다. Rust는 이를 허용할 수 없다.

변경할 수 없는 데이터 공유하기

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 어떻게 해결할 수 있을까?
 - 스레드 간 레퍼런스 공유를 지원하기엔 `spawn`이 너무 개방적인 게 아닌가 싶다.
실제로 '클로저'를 배울 때 이미 이런 경우를 살펴본 바 있다. (6주차 '훔치는 클로저' 참고)
 - 당시에는 `move` 클로저를 써서 데이터의 소유권을 새 스레드로 옮기는 식으로 해결했는데,
이번에는 같은 데이터를 써야 하는 스레드가 여러 개 있는 상황이라서 그 방법이 통하지 않는다.
 - 스레드마다 `glossary`를 통으로 `clone`하는 게 그나마 안전한 대안이지만, 덩치가 너무 커서 웬만하면 피하고 싶다.
다행히도 표준 라이브러리는 원자적인 레퍼런스 카운팅이라는 또 다른 대안을 제공한다. 바로 `Arc`다!

변경할 수 없는 데이터 공유하기

HSPACE Rust 특강
Rust Basic #14 - 동시성

- Arc를 사용해 문제를 해결해 보자!

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

fn process_files_in_parallel(filename: Vec<String>, glossary: Arc<GigabyteMap>) -> io::Result<()> {
    // ...

    for worklist in worklists {
        // Increase the reference count of the glossary instead of cloning it
        let glossary_for_child = glossary.clone();

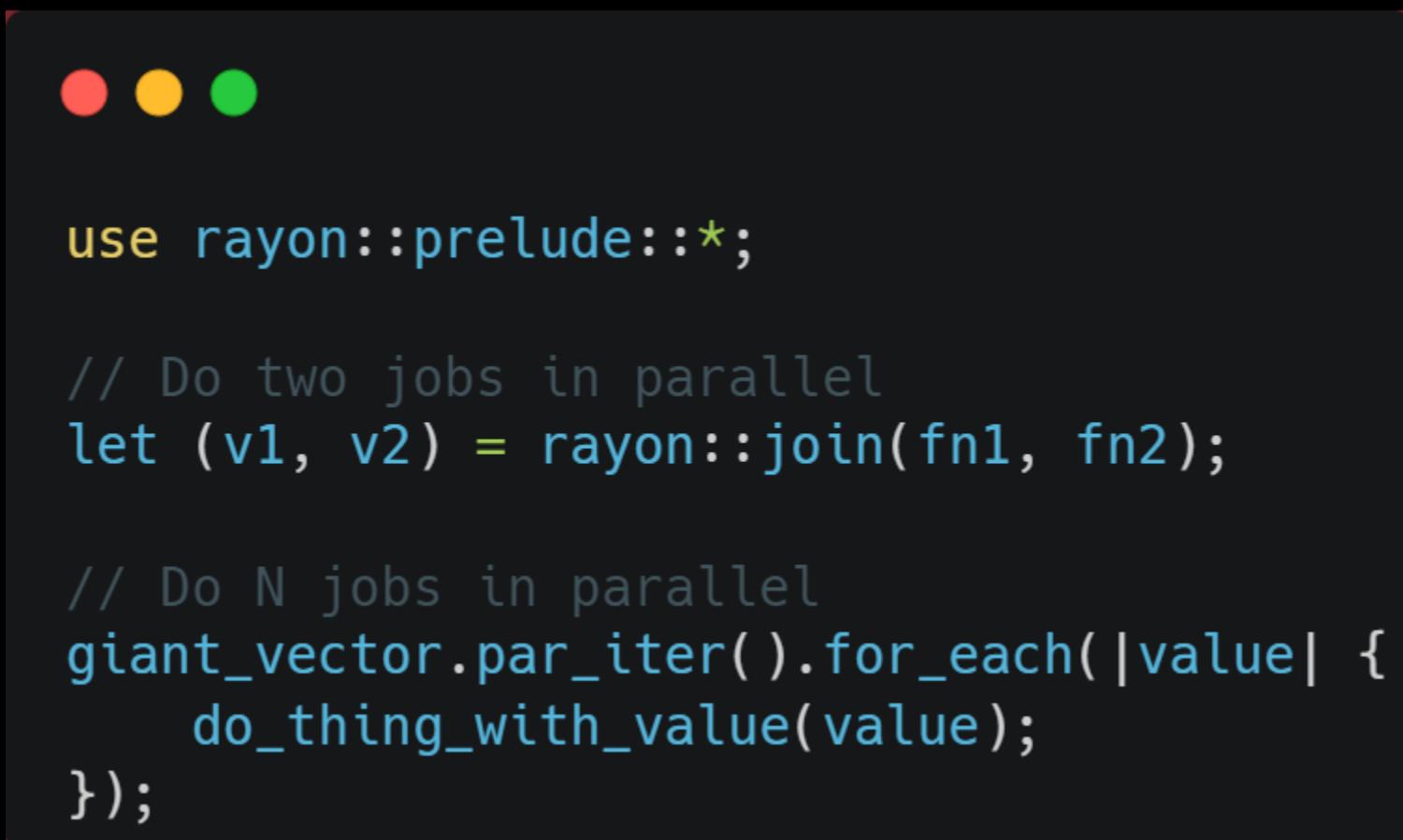
        thread_handles.push(thread::spawn(move || {
            process_files(worklist, &glossary_for_child)
        }));
    }

    // ...
}
```

Rayon 라이브러리

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 표준 라이브러리의 `spawn` 함수는 멀티 스레드에서 사용하는 기본 함수이지만 특별히 Fork-Join 병렬 처리를 위해서 설계되진 않았다.
- 그 대신 더 나은 Fork-Join API를 제공하는 Rayon이라는 라이브러리가 있다.



A screenshot of a terminal window with three colored window control buttons (red, yellow, green) at the top. The terminal displays the following Rust code:

```
use rayon::prelude::*;

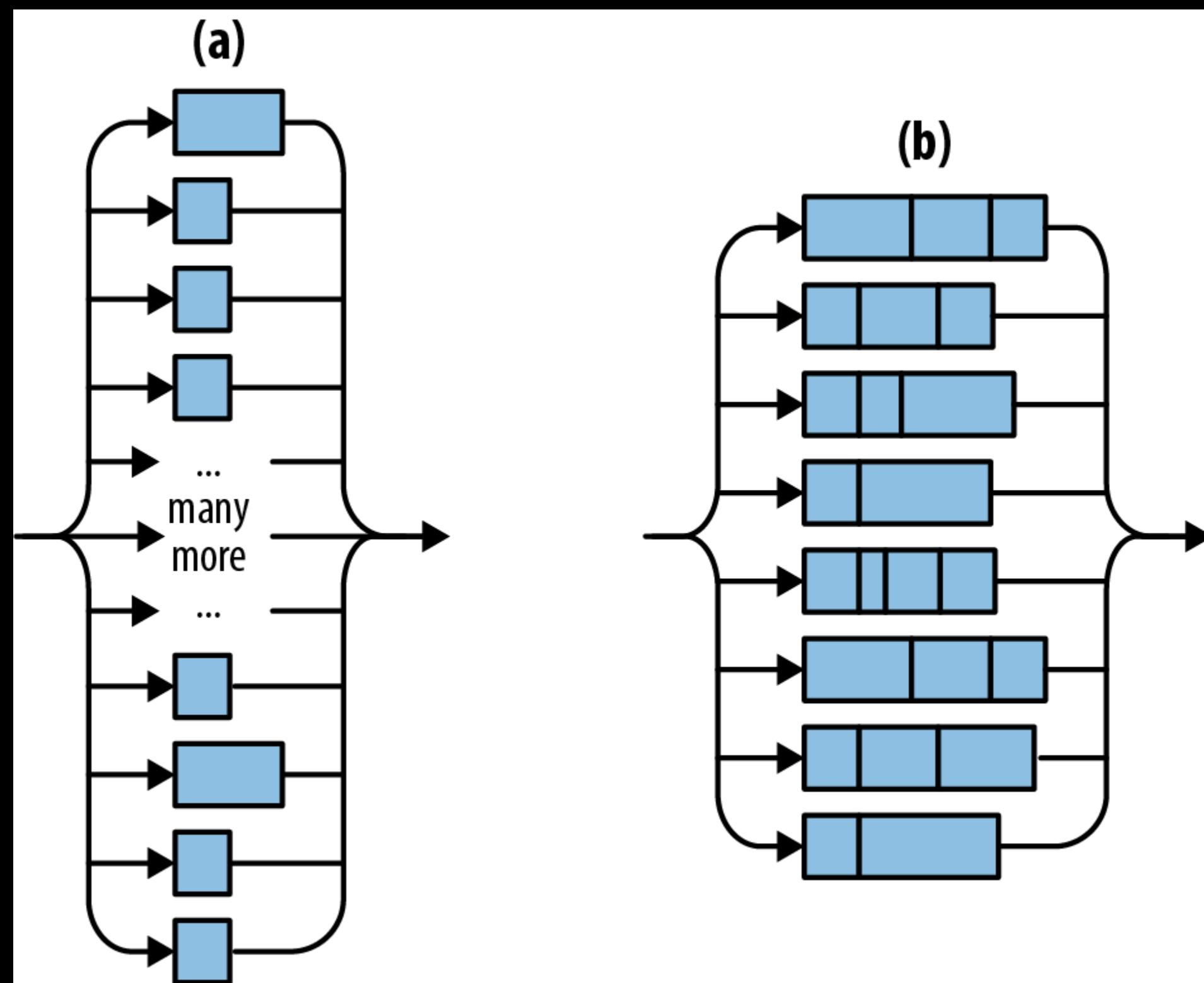
// Do two jobs in parallel
let (v1, v2) = rayon::join(fn1, fn2);

// Do N jobs in parallel
giant_vector.par_iter().for_each(|value| {
    do_thing_with_value(value);
});
```

Rayon 라이브러리

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 다음은 `giant_vector.par_iter().for_each(...)` 호출의 동작 방식을 보여준다.
 - (a) Rayon은 마치 벡터가 가진 요소마다 스레드를 하나씩 생성하는 것처럼 동작한다.
 - (b) Rayon은 이면에서 CPU 코어마다 워커 스레드를 하나씩 마련해 두고 있는데, 그쪽이 더 효율적이기 때문이다.
워커 스레드 풀은 프로그램의 모든 스레드가 공유하며, 수많은 작업이 동시에 들어오면 Rayon이 이들 작업을 나눈다.



Rayon 라이브러리

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 다음은 Rayon을 사용한 `process_files_in_parallel` 함수의 구현 코드다.

```
● ● ●

use rayon::prelude::*;

fn process_files_in_parallel(filenames: Vec<String>, glossary: &GigabyteMap) -> io::Result<()> {
    filenames
        .par_iter()
        .map(|filename| process_file(filename, glossary))
        .reduce_with(|r1, r2| if r1.is_err() { r1 } else { r2 })
        .unwrap_or(Ok(()))
}
```

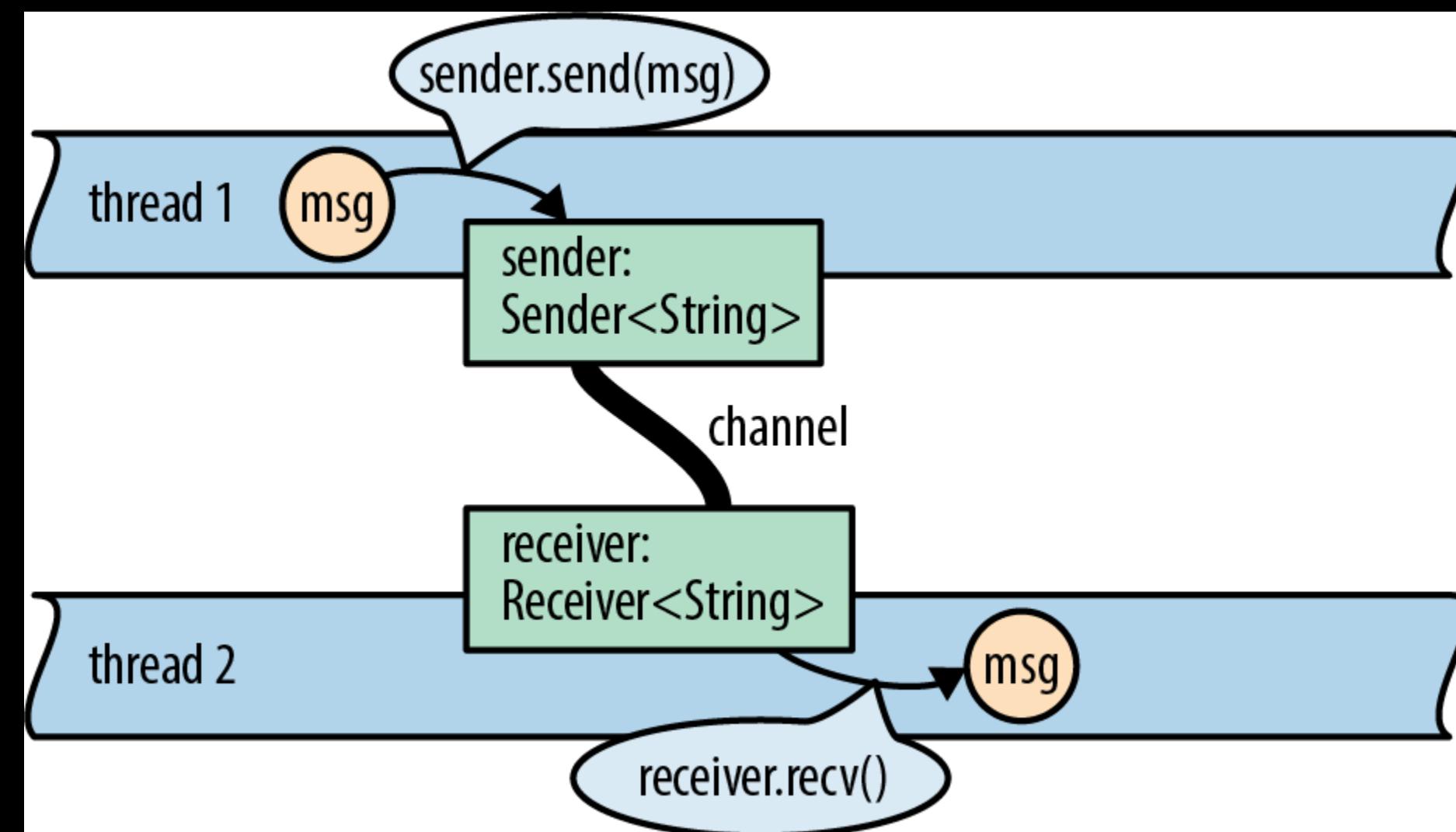
Rayon 라이브러리

HSPACE Rust 특강
Rust Basic #14 - 동시성

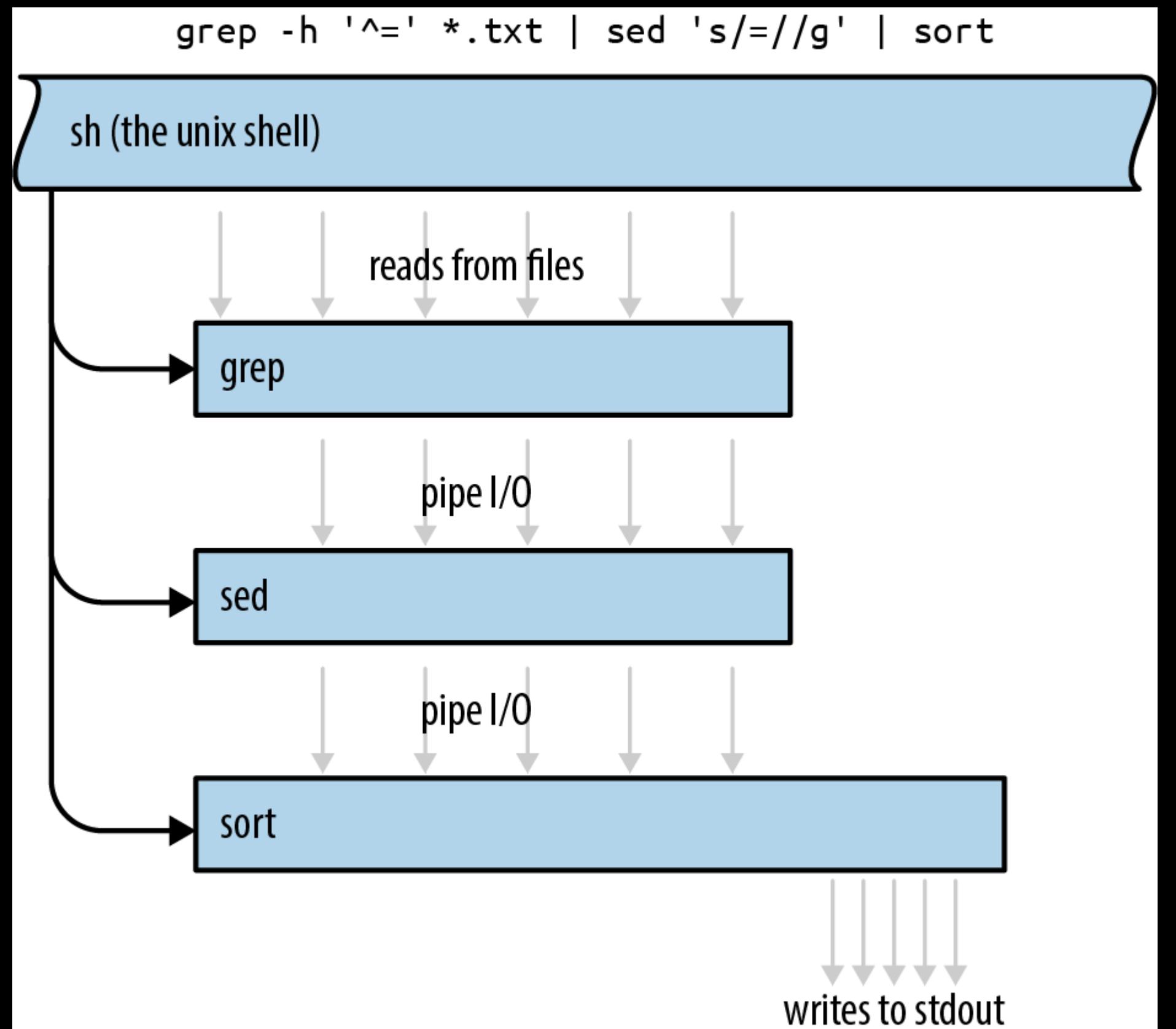
- Rayon의 특징
 - Rayon은 이면에서 작업 훔치기(Work-Stealing)이라고 하는 기법을 써서 스레드 간의 작업량을 동적으로 조절한다.
이 기법은 앞서 나온 spawn/join을 사용해 직접 작업을 미리 분할할 때보다 전체 CPU 사용율을 더 높게 가져갈 수 있다.
 - 또한 Rayon은 스레드 간 레퍼런스 공유를 지원한다.
이면에서 벌어지는 모든 병렬 처리는 reduce_with가 복귀하기 전에 반드시 끝난다. 클로저가 여러 스레드에서
호출되는 상황임에도 불구하고 glossary를 process_file에 넘길 수 있었던 이유가 바로 여기에 있다.

- **채널(Channel)**은 한 스레드에서 다른 스레드로 값을 보내기 위한 단방향 도관이다.

- 쉽게 말해서, 스레드 세이프한 큐라고 보면 된다.
- Unix의 파이프(Pipe)와 비슷한데 한쪽 끝은 데이터를 보낼 때 쓰고, 다른쪽 끝은 데이터를 받을 때 쓴다.
이 양쪽 끝은 통상 서로 다른 두 스레드가 소유한다.
- Unix의 파이프는 바이트 열을 보내기 위한 것인 반면, 채널은 Rust 값을 보내기 위한 것이다.
- `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 бл록된다.



- **채널(Channel)**은 한 스레드에서 다른 스레드로 값을 보내기 위한 단방향 도관이다.
 - 채널을 쓰면 스레드가 서로 값을 주고 받으며 통신할 수 있다.
잠금이나 공유된 메모리를 쓰지 않고도 스레드가 서로 협업할 수 있게 해주는 아주 단순한 장치다.
 - 사실 채널은 새로운 기술이 아니다.
 - 얼랭(Erlang)은 30년 전부터 격리된 프로세스와 메시지 전달을 사용해 왔다.
 - Unix의 파이프가 사용된 지는 거의 50년이 다 되어 간다.



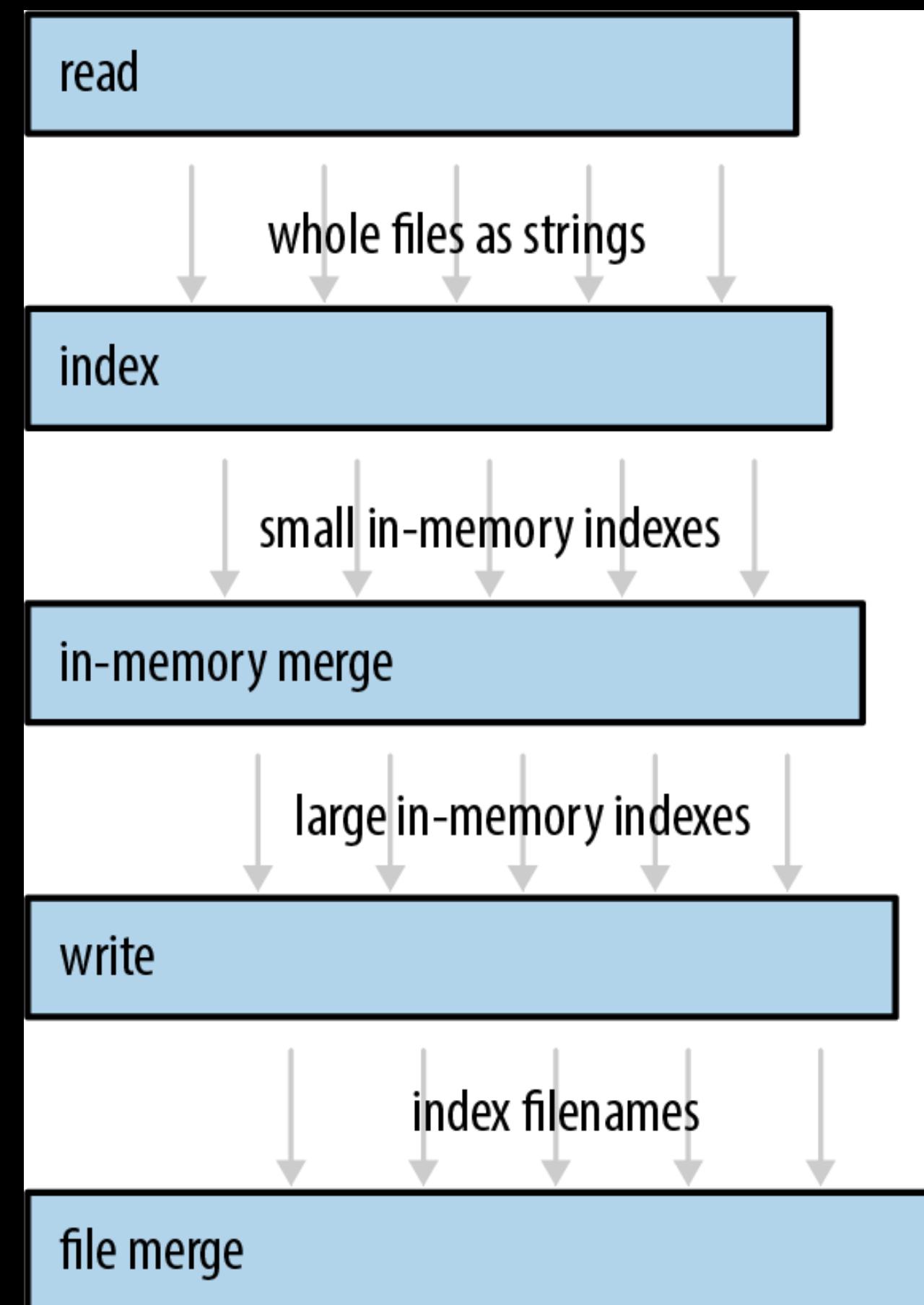
값 보내기

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 지금부터 검색 엔진의 핵심 요소 중 하나인 역색인(Inverted Index)을 생성하는 동시성 프로그램을 만들어 보려고 한다.

- 모든 검색 엔진은 특정 문서 집합을 대상으로 동작한다.
- 역색인은 어떤 단어가 어디에 등장하는지 알려 주는 데이터베이스다.
- 여기서는 스레드와 채널을 쓰는 코드 부분만 살펴 본다.

전체 프로그램 코드는 “2 - Example” 폴더를 참고하기 바란다.



- 이 프로그램은 다음과 같은 형태의 파이프라인으로 구성되어 있다.
 - 파이프라인은 채널을 사용하는 여러 방법 중 하나에 불과하지만, 이를 사용하면 이미 존재하는 싱글 스레드 프로그램에 동시성을 손쉽게 도입할 수 있다.
 - 이 파이프라인은 총 5개의 스레드를 사용하며 각자 고유한 작업을 수행한다. 각 스레드는 프로그램의 수명 동안 결과를 지속적으로 산출한다.
 - 예를 들어, 첫 번째 스레드는 디스크에 있는 소스 문서를 하나씩 메모리로 읽어오는 일만 한다. 이 단계에서는 결과로 문서마다 긴 String이 하나씩 나오기 때문에 이 스레드는 String 채널을 통해서 다음 스레드와 연결된다.

값 보내기

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 이 프로그램은 파일을 읽는 스레드를 생성하는 것으로 시작한다.

```
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 보내기

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 편의상 이전 코드를 다음처럼 생긴 함수로 감싸서 사용한다.
이 함수는 (아직 쓰지 않은) receiver와 새 스레드의 JoinHandle을 한꺼번에 반환한다.

```
● ● ●

fn start_file_reader_thread(
    documents: Vec<PathBuf>,
) -> (mpsc::Receiver<String>, thread::JoinHandle<io::Result<()>>) {
    let (sender, receiver) = mpsc::channel();

    let handle = thread::spawn(move || {
        // ...
    });

    (receiver, handle)
}
```

- 값을 보내는 반복문을 실행하는 스레드가 준비되었으니,
이제 `receiver.recv()`를 호출하는 반복문을 실행하는 두 번째 스레드를 만들 차례다.
- 참고로 다음 두 반복문이 하는 일은 똑같다.

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

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

- 이제 파이프라인의 두 번째 단계를 위한 코드를 작성해 보자.

```
● ● ●

fn start_file_indexing_thread(
    texts: mpsc::Receiver<String>,
) -> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>) {
    let (sender, receiver) = mpsc::channel();

    let handle = thread::spawn(move || {
        for (doc_id, text) in texts.into_iter().enumerate() {
            let index = InMemoryIndex::from_single_document(doc_id, text);
            if sender.send(index).is_err() {
                break;
            }
        }
    });
    (receiver, handle)
}
```

파이프라인 실행하기

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 나머지 세 단계의 설계는 유사하다.
 - 각 단계는 앞 단계가 생성한 Receiver를 소비한다.
 - 나머지 파이프라인의 목표는 작은 색인들을 전부 가져다가 디스크에 있는 커다란 색인 파일 하나에 모아 담는 것이다.
이를 위해 생각해볼 수 있는 가장 빠른 방법은 일을 세 단계로 나누는 것이다.
 - 단계 3 : 먼저 색인을 메모리에 최대한 모아 담는다.



```
fn start_in_memory_merge_thread(
    file_indexes: mpsc::Receiver<InMemoryIndex>,
) -> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>)
```

파이프라인 실행하기

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 나머지 세 단계의 설계는 유사하다.
 - 각 단계는 앞 단계가 생성한 Receiver를 소비한다.
 - 나머지 파이프라인의 목표는 작은 색인들을 전부 가져다가 디스크에 있는 커다란 색인 파일 하나에 모아 담는 것이다. 이를 위해 생각해볼 수 있는 가장 빠른 방법은 일을 세 단계로 나누는 것이다.
 - 단계 4 : 이렇게 만들어진 커다란 색인을 디스크에 기록한다.

```
fn start_index_writer_thread(  
    big_indexes: mpsc::Receiver<InMemoryIndex>,  
    output_dir: &Path,  
) -> (mpsc::Receiver<PathBuf>, thread::JoinHandle<io::Result<()>>)
```

파이프라인 실행하기

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 나머지 세 단계의 설계는 유사하다.
 - 각 단계는 앞 단계가 생성한 Receiver를 소비한다.
 - 나머지 파이프라인의 목표는 작은 색인들을 전부 가져다가 디스크에 있는 커다란 색인 파일 하나에 모아 담는 것이다. 이를 위해 생각해볼 수 있는 가장 빠른 방법은 일을 세 단계로 나누는 것이다.
 - 단계 5 : 커다란 파일이 여러 개 생기면 이들을 파일 기반의 병합 알고리즘을 써서 하나로 합친다.

```
● ● ●  
fn merge_index_files(files: mpsc::Receiver<PathBuf>, output_dir: &Path) -> io::Result<()>
```

파이프라인 실행하기

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 이제 스레드를 띄우고 오류를 검사하는 코드를 살펴 보자.

```
● ● ●

fn run_pipeline(documents: Vec<PathBuf>, output_dir: PathBuf) -> io::Result<()> {
    // Launch all five stages of the pipeline
    let (texts, h1) = start_file_reader_thread(documents);
    let (pints, h2) = start_file_indexing_thread(texts);
    let (gallons, h3) = start_in_memory_merge_thread(pints);
    let (files, h4) = start_index_writer_thread(gallons, &output_dir);
    let result = merge_index_files(files, &output_dir);

    // Wait for threads to finish, holding on to any errors that they encounter
    let r1 = h1.join().unwrap();
    h2.join().unwrap();
    h3.join().unwrap();
    let r4 = h4.join().unwrap();

    // Return the first error encountered, if any
    // As it happens, h2 and h3 can't fail:
    // those threads are pure in-memory data processing
    r1?;
    r4?;
    result
}
```

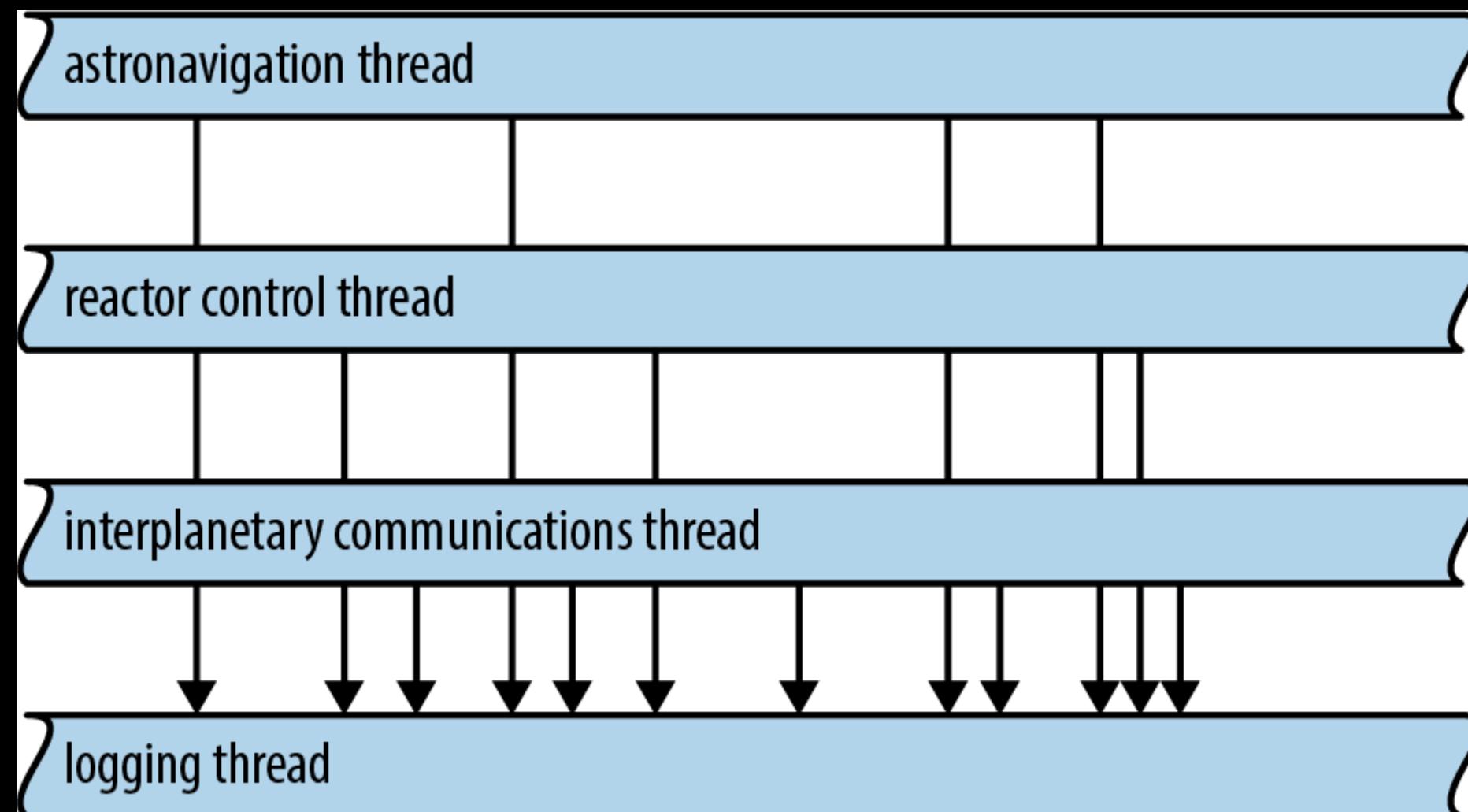
채널의 기능과 성능

HSPACE Rust 특강
Rust Basic #14 - 동시성

- `std::sync::mpsc`에서 `mpsc`는 무엇일까?

바로 멀티 프로듀서, 싱글 컨슈머(Multi-Producer, Single-Consumer)의 약자로,
Rust의 채널이 제공하는 통신의 종류를 한마디로 설명해 준다.

- 이전 프로그램에 있는 채널은 보내는 쪽 하나에서 받는 쪽 하나로 값을 실어 나른다. 이런 경우는 꽤 흔하다.
- 그러나 Rust의 채널은 보내는 쪽이 여럿인 경우도 지원한다.



채널의 기능과 성능

HSPACE Rust 특강
Rust Basic #14 - 동시성

- std::sync::mpsc에서 mpsc는 무엇일까?

바로 멀티 프로듀서, 싱글 컨슈머(Multi-Producer, Single-Consumer)의 약자로,
Rust의 채널이 제공하는 통신의 종류를 한마디로 설명해 준다.

- 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. 이는 뒤에서 살펴볼 예정이다.

채널의 기능과 성능

HSPACE Rust 특강
Rust Basic #14 - 동시성

- Rust의 채널은 아주 꼼꼼하게 최적화되어 있다.
 - Rust는 처음 채널을 만들 때 특별한 “일회성” 큐(Queue) 구현을 쓴다. 이 구현은 채널을 통해서 보내는 객체가 하나뿐일 때 오버헤드를 최소화하도록 설계했다.
 - 이 채널에 대고 두 번째 값을 보내면 Rust는 다른 큐 구현으로 전환한다. 그리고는 할당 오버헤드를 최소화하면서 많은 값을 전송할 수 있는 채널로 탈바꿈해 비교적 오랫동안 여기에 정착한다.
 - 그러나 Sender를 복제하면 Rust는 여러 스레드가 동시에 값을 보내려고 해도 안전한 또 다른 구현으로 옮겨가야 한다.
 - 그러나 이 세 가지 구현 중에서 가장 느리다고 하는 게 Lock-Free 큐라서, 값을 보내거나 받을 때 드는 비용이 기껏해야 원자적인 연산 몇 번에 힘 할당과 이동 자체가 더해지는 수준이다.
 - 시스템 호출이 필요한 순간은 큐가 비어서 받는 스레드가 스스로 대기 상태에 들어가야 할 때뿐이다. 물론, 이 경우에는 어찌 됐든 채널의 통행량이 최대치를 찍지 않는다.

채널의 기능과 성능

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 하지만 최적화가 전부 작동하더라도 채널 성능 측면에서 저지르기 쉬운 실수가 하나 있다.
 - 바로 값을 받아서 처리할 수 있는 속도보다 더 빨리 보내는 것이다.
 - 이렇게 하면 채널 안에 밀린 값이 점점 쌓이게 된다.
예를 들어, 만들었던 프로그램은 파일을 읽는 스레드(단계 1)에서 파일을 불러오는 속도가 파일 색인을 만드는 스레드(단계 2)에서 색인을 만드는 속도보다 훨씬 더 빠를 수 있다.
그 결과 디스크에서 읽어 온 수백 메가바이트의 원시 데이터가 큐에 한꺼번에 쌓이게 된다.
- 이런 식으로 잘못 작동하게 그냥 놔두면 메모리가 낭비되고 지역성(Locality)이 훼손된다.
설상가상으로 보내는 스레드가 계속 실행되면서 처리하지도 못할 값을 보내느라 받는 쪽이 절실히 필요로 하는 CPU와 기타 시스템 리소스를 다 써버리게 된다.

채널의 기능과 성능

HSPACE Rust 특강
Rust Basic #14 - 동시성

- Rust는 이 부분에서 다시 Unix 파일처럼 작동한다.
 - Unix는 빠르게 보내는 쪽의 속도를 강제로 낮추는 일종의 **배압(Backpressure)**을 제공하기 위해서 우아한 꼼수를 쓴다.
 - Unix 시스템에서 각 파일은 크기가 고정되어 있으며, 프로세스가 일시적으로 가득 찬 파일에 기록하려고 하면 시스템은 파일에 공간이 생길 때까지 프로세스를 그냥 블록시킨다.
 - Rust는 여기에 상응하는 **동기 채널(Synchronous Channel)**이라는 기능이 있다.
 - 동기 채널은 일반 채널과 똑같지만 생성할 때 줄 수 있는 값의 개수를 지정한다는 점이 다르다. 동기 채널의 경우에는 `sender.send(value)`가 블록될 가능성이 있는 작업이다.
 - 어쨌든 핵심 아이디어는 블로킹이 항상 나쁜 게 아니라는 점이다. 만들었던 프로그램에서 `start_file_reader_thread`에 있는 `channel`을 32개의 값을 줄 수 있는 `sync_channel`로 바꾸면, 처리량은 그대로 유지하면서 메모리 사용량을 줄어드는 효과를 볼 수 있다.

스레드 안전성 : Send와 Sync

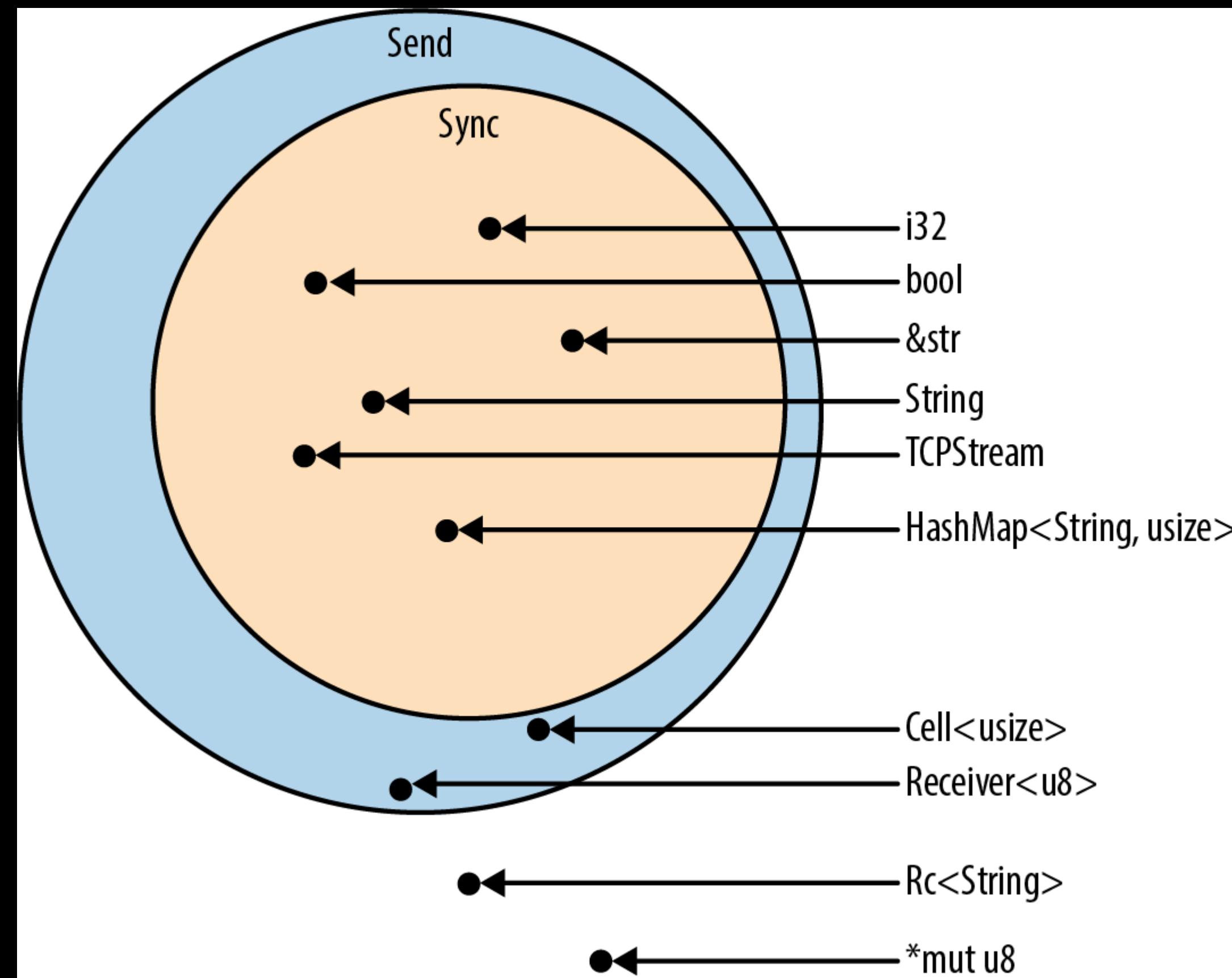
HSPACE Rust 특강
Rust Basic #14 - 동시성

- 지금까지 마치 모든 값을 스레드 간에 자유롭게 옮기고 공유할 수 있는 것처럼 행동해왔다.
- Rust에서 스레드 안전성과 관련된 전체 이야기는 두 가지 기본 제공 트레이트인 `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.
- 여기서 안전하다는 건 데이터 경합과 정의되지 않은 동작이 없다는 뜻이다.
예를 들어, 앞에서 봤던 `process_files_in_parallel` 함수에서 클로저를 써서 부모 스레드에 있는 `Vec<String>`을 각 자식 스레드에 넘겼다. 그땐 지적하지 않았지만 이 말은 부모 스레드에서 할당된 벡터와 그 안에 있는 문자열이 자식 스레드에서 해제된다는 뜻이다. `Vec<String>`이 `Send`를 구현하고 있다는 사실은 그렇게 해도 된다는 API의 약속이다. `Vec`과 `String`이 내부적으로 쓰는 Allocator는 스레드 세이프하다.

스레드 안전성 : Send와 Sync

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 대부분의 타입은 Send면서 Sync다. 프로그램에 있는 구조체나 열거체에 `#[derive]`를 붙이지 않아도 Rust가 Send와 Sync를 알아서 구현해 준다.



스레드 안전성 : Send와 Sync

HSPACE Rust 특강
Rust Basic #14 - 동시성

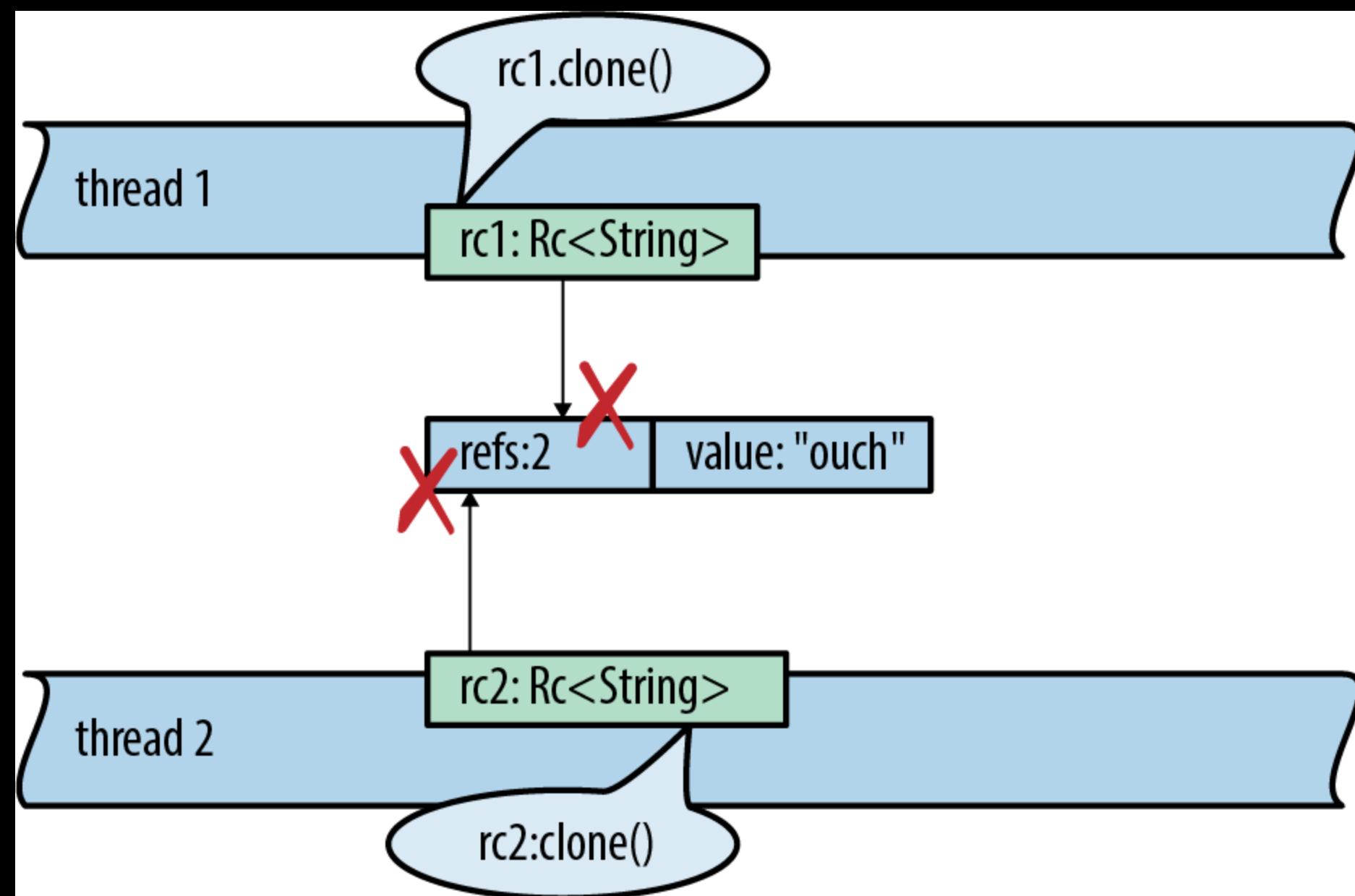
- 예외 사항
 - Send지만 Sync가 아닌 타입도 있는데, 이는 보통 의도적인 결정일 때가 많다.
`mpsc::Receiver`는 이런 특성을 이용해서 `mpsc` 채널의 받는 쪽을 한 번에 한 스레드에서만 쓸 수 있게 보장한다.
 - 소수지만 Send도 아니고 Sync도 아닌 타입은 대부분 스레드 세이프하지 않은 방식으로 가변성을 이용한다.
예를 들어, 레퍼런스 카운트를 쓰는 스마트 포인터 타입 `std::rc::Rc<T>`를 보자.

스레드 안전성 : Send와 Sync

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 예외 사항

- `Rc<String>`이 `Sync`여서 `Rc` 하나를 공유된 레퍼런스를 통해서 스레드 간에 공유할 수 있다면 무슨 일이 벌어질까?
두 스레드가 동시에 `Rc`를 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.



스레드 안전성 : Send와 Sync

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 다음 코드는 데이터 경합을 유발한다.



```
● ● ●

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();
}
```

스레드 안전성 : Send와 Sync

HSPACE Rust 특강
Rust Basic #14 - 동시성

- Rust는 컴파일을 거부하고 자세한 오류 메시지를 던져 준다.

```
● ● ●

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`
```

- 거의 모든 이터레이터를 연결해 쓸 수 있는 채널
 - 앞에서 만들었던 역색인 작성기는 파이프라인 형태로 만들어졌다.
코드는 깔끔한 편이지만 직접 채널을 설정하고 스레드를 띄워야 했다.
좀 더 편하게 만들 수 없을까? 다음처럼 이터레이터 파이프라인 형태로 작성할 수 있다면 좋겠다.

```
● ● ●

documents.into_iter()
    .map(read_whole_file)
    .errors_to(error_sender)      // Filter out error results
    .off_thread()                 // Spawn a thread for the above work
    .map(make_single_file_index)
    .off_thread();                // Spawn another thread for stage 2
    ...
```

- 거의 모든 이터레이터를 연결해 쓸 수 있는 채널
 - 트레잇을 쓰면 표준 라이브러리 타입에 메소드를 추가할 수 있으므로 실제로 이렇게 할 수 있다.
먼저, 원하는 메소드를 선언하는 트레잇을 작성하는 것으로 시작하자.

```
● ● ●  
  
use std::sync::mpsc;  
  
pub trait OffThreadExt: Iterator {  
    // Transform this iterator into an off-thread iterator:  
    // the `next()` calls happen on a separate worker thread  
    // so the iterator and the body of your loop run concurrently  
    fn off_thread(self) -> mpsc::IntoIter<Self::Item>;  
}
```

- 거의 모든 이터레이터를 연결해 쓸 수 있는 채널
 - 그런 다음 이 트레잇을 Iterator 타입에 대해서 구현한다.
`mpsc::Receiver`가 Iterator에 대해 구현되어 있어서 간단히 끝낼 수 있다.

```
use std::thread;

impl<T> OffThreadExt for T
where
    T: Iterator + Send + 'static,
    T::Item: Send + 'static,
{
    fn off_thread(self) -> mpsc::IntoIter<Self::Item> {
        // Create a channel to transfer items from the worker thread
        let (sender, receiver) = mpsc::sync_channel(1024);

        // Move this iterator to a new worker thread and run it there
        thread::spawn(move || {
            for item in self {
                if sender.send(item).is_err() {
                    break;
                }
            }
        });
        // Return an iterator that pulls values from the channel
        receiver.into_iter()
    }
}
```

변경할 수 있는 공유된 상태

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 양치식물 시뮬레이션 소프트웨어 업데이트

- 이번에는 8명의 플레이어가 가상의 쥐라기 시대를 배경으로 현실과 거의 똑같은 기간 동안 양치식물을 키우며 경쟁하는 멀티 플레이어 실시간 전략 게임을 만드는 중이라고 하자.
- 이 게임의 서버는 많은 스레드에 요청을 쏟아붓는 대규모 병렬 애플리케이션이다.
이들 스레드를 어떤 식으로 조정해야 8명의 플레이어가 곧바로 게임을 시작할 수 있을까?
- 여기서 풀어야 할 문제는 많은 스레드가 게임에 참여하기 위해서 기다리는 플레이어들의 공유된 목록에 접근해야 한다는 점이다.
이 데이터는 모든 스레드가 변경할 수 있어야 하고 또 공유할 수 있어야 한다.
Rust가 변경할 수 있는 공유된 상태를 갖지 않는다면 어떻게 해야 좋을까?

변경할 수 있는 공유된 상태

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 양치식물 시뮬레이션 소프트웨어 업데이트
 - 1) 해당 목록을 관리하는 일만 전담하는 새 스레드를 만드는 방법
이때 다른 스레드는 채널을 통해서 이 스레드와 통신하게 된다.
물론 이 방법은 스레드를 하나 더 쓰는 셈이니, 이로 인한 약간의 운영체제 오버헤드는 감수해야 한다.
 - 2) Rust가 변경할 수 있는 데이터를 안전하게 공유할 수 있도록 제공하는 도구를 쓰는 방법
스레드를 다뤄 본 시스템 프로그래머라면 누구나 잘 알고 있을 법한 저수준 기본 요소를 사용하면 된다.
- 지금부터 뮤텍스, 읽기/쓰기 락, 조건 변수, 원자적인 정수를 다룬다.
끝으로 Rust에서 변경할 수 있는 전역 변수를 구현하는 법도 살펴 본다.

뮤텍스란?

HSPACE Rust 특강
Rust Basic #14 - 동시성

- **뮤텍스(Mutex)** 또는 **락(Lock)**은 특정 데이터를 여러 스레드가 필요로 할 때 강제로 번갈아 가며 접근하도록 만들기 위해서 쓴다.
- Rust의 Mutex는 잠시 후에 알아보도록 하고, 먼저 다른 언어에서는 어떤 식으로 다루는지 알아 보자.
- C++에서는 뮤텍스를 다음과 같이 쓴다.

```
// C++ code, not Rust
void FernEmpireApp::JoinWaitingList(PlayerId player)
{
    mutex.Acquire();

    waitingList.push_back(player);

    // Start a game if we have enough players waiting
    if (waitingList.size() >= GAME_SIZE)
    {
        vector<PlayerId> players;
        waitingList.swap(players);
        StartGame(players);
    }

    mutex.Release();
}
```

뮤텍스란?

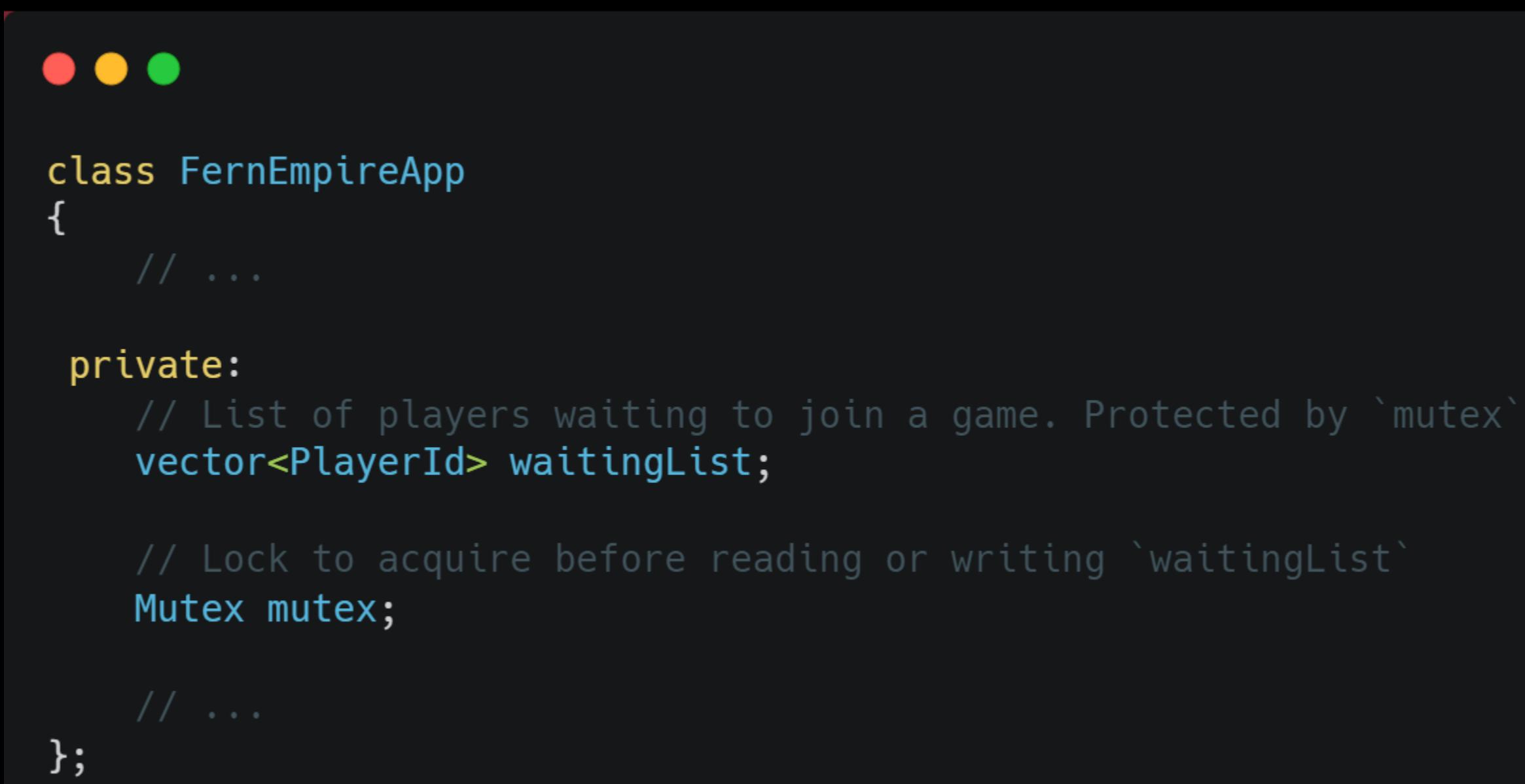
HSPACE Rust 특강
Rust Basic #14 - 동시성

- 뮤텍스가 유용한 데는 몇 가지 이유가 있다.
 - 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 (종합적으로 볼 때) 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. 다른 스레드가 실행 중에 데이터를 수정하더라도 작동하는 코드를 작성한다고 생각해 보자. 또 그걸 디버깅한다고 생각해 보자. 아마 프로그램이 귀신에 쓴 게 아닐까 싶을 것이다.
 - 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.

뮤텍스란?

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 하지만 대부분의 언어에서 뮤텍스는 엉망진창이 되기 쉽다.
 - C++에서는 대부분의 언어와 마찬가지로 데이터와 락이 별개의 객체다. 이론적으로 “모든 스레드는 데이터를 건드리기 전에 반드시 뮤텍스를 획득하세요”라고 주석으로 남겨 두는 것 외에는 이를 제대로 설명할 방법이 없다.
 - 그러나 아무리 주석을 잘 남겨 둬도 컴파일러는 안전한 접근을 강제할 수 없다. 일부 코드에서 깜빡하고 뮤텍스를 획득하지 않으면 정의되지 않은 동작이 발생한다. 실제로 이 말은 재현하고 고치기가 극도로 어려운 버그란 뜻이다.



```
● ● ●

class FernEmpireApp
{
    // ...

private:
    // List of players waiting to join a game. Protected by `mutex`
    vector<PlayerId> waitingList;

    // Lock to acquire before reading or writing `waitingList`
    Mutex mutex;

    // ...
};
```

- 이제 Rust로 구현한 대기자 목록을 살펴 보자.
 - FernEmpire 게임의 서버에서는 각 플레이어가 고유한 ID를 갖는다.

```
● ● ●  
type PlayerId = u32;
```

- 대기자 명단은 플레이어의 컬렉션에 불과하다.

```
● ● ●  
const GAME_SIZE: usize = 8;  
  
// A waiting list never grows to more than GAME_SIZE players  
type WaitingList = Vec<PlayerId>;
```

Mutex<T>

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 이제 Rust로 구현한 대기자 목록을 살펴 보자.
 - 대기자 명단은 FernEmpireApp의 필드로 저장된다.
FernEmpireApp은 서버가 시작하는 과정에서 Arc 안에 생성되는 싱글톤이다.
각 스레드는 이를 가리키는 Arc를 갖는다.
여기에는 공유된 구성을 비롯해 프로그램이 필요로 하는 온갖 잡동사니가 전부 들어가 있는데, 대부분은 읽기 전용이다.
대기자 명단은 변경할 수 있는 채로 공유되기 때문에 반드시 Mutex로 보호해야 한다.

```
● ● ●  
  
use std::sync::Arc;  
  
let app = Arc::new(FernEmpireApp {  
    // ...  
    waiting_list: Mutex::new(vec![]),  
    // ...  
});
```

- C++와 달리 Rust는 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, 여기서는 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.

Mutex<T>

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 이제 join_waiting_list 메소드를 Mutex를 써서 구현해 보자.

```
● ● ●

impl FernEmpireApp {
    /// Add a player to the waiting list for the next game
    /// Start a new game immediately if enough players are waiting
    fn join_waiting_list(&self, player: PlayerId) {
        // Lock the mutex and gain access to the data inside
        // The scope of `guard` is a critical section
        let mut guard = self.waiting_list.lock().unwrap();

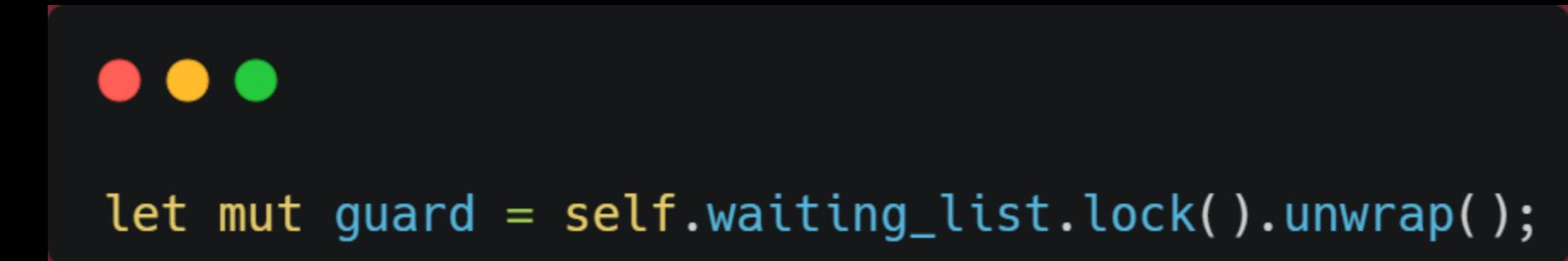
        // Now do the game logic
        guard.push(player);

        if guard.len() == GAME_SIZE {
            let players = guard.split_off(0);
            self.start_game(players);
        }
    }
}
```

mut와 Mutex

HSPACE Rust 특강
Rust Basic #14 - 동시성

- join_waiting_list 메소드를 보면 이상한 점이 하나 있다.
 - 이 메소드의 첫 번째 매개 변수는 &mut self가 아닌 &self이다.



```
let mut guard = self.waiting_list.lock().unwrap();
```

- Vec<PlayerId> 타입에 대해 push 메소드를 호출하고 있다. 이 메소드는 &mut self를 요구한다.
- 그런데 앞의 코드는 컴파일도 되고 실행도 된다. 대체 무슨 일이 벌어진 걸까?



```
if guard.len() == GAME_SIZE {  
    let players = guard.split_off(0);  
  
    // Don't keep the list locked while starting a game  
    drop(guard);  
  
    self.start_game(players);  
}
```

- Rust에서 `&mut`는 **배타적 접근(Exclusive Access)**을 의미하고, `&`는 **공유된 접근(Shared Access)**을 의미한다.
- 우리는 부모에서 자식으로, 또 컨테이너에서 요소로 `&mut` 접근 권한을 넘기는 방식에 익숙하다. 그러나 보니 이를테면 발사할 `starships`의 `&mut` 레퍼런스를 손에 쥐고 있을 때만 `starships[id].engine`에 대고 `&mut self` 메소드를 호출할 수 있다고 여긴다. Rust는 일반적으로 부모에 대한 배타적 접근 권한을 가지고 있지 않으면 자식에 대한 배타적 접근 권한을 가지게 만들 방법이 없으므로 이 익숙한 방식을 기본 동작으로 삼는다.
- 그러나 Mutex는 이에 대한 해결책을 가지고 있는데, 락이 바로 그것이다. 사실 뮤텍스는 많은 스레드가 Mutex 자체에 대한 공유된(`mut`가 아닌) 접근 권한을 가질 수 있는 상황에서도, 락을 써서 자기가 가진 데이터에 대한 배타적(`mut`) 접근 권한을 제공하는 방법에 지나지 않는다.
- Rust의 타입 시스템은 Mutex가 무슨 일을 하는지 말해 준다. Mutex는 일반적으로 Rust 컴파일러가 컴파일 시점에 정적으로 시행하는 배타적 접근 규칙을 동적으로 처리한다.

뮤텍스만 고집하는 그대에게

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 앞서 우리는 뮤텍스에 대해 알아보기 전에 동시성에 관한 몇 가지 접근 방식을 살펴봤었다.
 - C++에서 넘어온 사람이라면 접근 방식들이 이상하리만큼 쓰기도 쉽고 실수할 여지도 적어 보였을지 모르겠는데, 그건 우연이 아니다. 이들은 동시성 프로그래밍의 가장 혼란스러운 측면에 맞서 강력한 보장을 제공하도록 설계됐다.
 - Fork-Join 병렬 처리만 쓰는 프로그램은 결정성(Deterministic)을 띠며, 교착 상태에 빠질 수 없다. 채널을 쓰는 프로그램은 거의 다 잘 작동한다.
 - 앞서 봤던 색인 작성기처럼 채널을 파이프라인 구축에만 쓰는 프로그램은 결정성을 띤다. 즉, 메시지 전달 타이밍은 다를 수 있어도 결과에는 영향을 주지 않는다.

뮤텍스만 고집하는 그대에게

HSPACE Rust 특강
Rust Basic #14 - 동시성

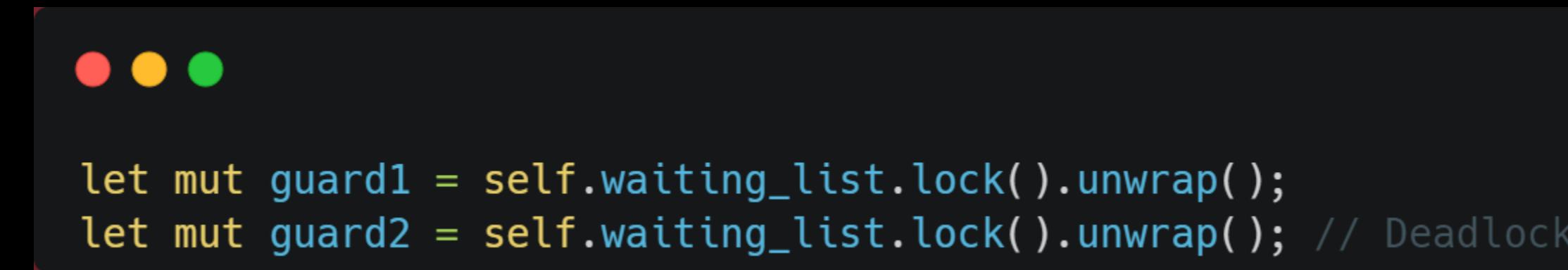
- 앞서 우리는 뮤텍스에 대해 알아보기 전에 동시성에 관한 몇 가지 접근 방식을 살펴봤었다.
 - Rust의 Mutex가 전보다 뮤텍스를 더 체계적이고 현명하게 쓸 수 있도록 설계된 건 거의 확실하다. 그러나 잠시 멈춰서서 Rust의 안전성 보장이 도움을 줄 수 있는 것과 없는 것을 두고 한 번쯤 짚어볼 필요가 있다.
 - 안전한 Rust 코드는 데이터 경합(Data Race), 즉 여러 스레드가 동시에 같은 메모리를 읽고 쓰는 바람에 무의미한 결과를 산출하는 이런 유형의 버그를 유발할 수 없다. 데이터 경합이 늘 버그를 유발한다는 점과 실제로 멀티 스레드 프로그램에서 드물지 않게 있는 일이란 걸 감안할 때 그 점은 아주 훌륭하다고 볼 수 있다.

뮤텍스만 고집하는 그대에게

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 뮤텍스를 쓰는 스레드는 Rust가 대신 해결해 주지 않는 다른 몇 가지 문제에 휩싸이기 쉽다.
 - 유효한 Rust 프로그램에는 데이터 경합이 있을 수 없지만 여전히 다른 **경합 조건(Race Condition)**, 즉 프로그램의 동작이 스레드 간의 타이밍에 따라 달라지는 바람에 실행할 때마다 결과가 달라지는 상황이 있을 수 있다. 어떤 경합 조건은 그냥 무시하고 넘어갈 만한 수준이지만, 경우에 따라서는 결과가 매번 제멋대로라 아주 고치기 힘든 버그가 되기도 한다. 뮤텍스를 구조화되지 않은 방식으로 쓰면 경합 조건이 생긴다. 이걸 그냥 무시하고 넘어갈 만한 수준으로 만드는 건 여러분의 몫이다.
 - 변경할 수 있는 공유된 상태 역시 프로그램 설계에 영향을 미친다. 채널은 코드에서 추상의 경계로 작용하기 때문에 테스트를 위해서 격리된 구성 요소로 분리하기가 쉬운 반면, 뮤텍스는 “그냥 메소드 하나 더 넣지 뭐” 식의 해결책을 부추기기 때문에 밀접한 연관이 있는 코드들이 하나의 거대한 덩어리를 이룰 가능성이 있다.
 - 끝으로 뮤텍스는 첫인상과 달리 그렇게 단순하지만은 않은데, 이건 지금부터 살펴 보려고 한다.

- 스레드가 자신이 이미 쥐고 있는 락을 획득하려 들면 교착 상태에 빠질 수 있다.



A screenshot of a terminal window with three colored dots (red, yellow, green) at the top left. The terminal displays the following code:

```
let mut guard1 = self.waiting_list.lock().unwrap();
let mut guard2 = self.waiting_list.lock().unwrap(); // Deadlock
```

- 첫 번째 `self.waiting_list.lock()` 호출이 락을 획득하는 데 성공했다고 하자.
이때 두 번째 호출은 누군가 이미 락을 획득해 있다고 판단하고, 락 해제가 일어날 때까지 대기 중인 채로 бл록된다.
그런데 여기서 락을 획득한 장본인이 바로 대기 중인 스레드 자신이라서 이 기다림의 끝은 영영 오지 않는다.
달리 말하면 Mutex의 락은 재귀 락이 아니다.
- 앞의 코드는 누가 봐도 버그다. 실제 프로그램에서는 서로 다른 두 메소드가 각자 `lock()` 호출을 포함하고 있는 상태에서 어느 한쪽이 다른 한쪽을 호출할 때 이 같은 일이 벌어질 수 있다. 이런 문제는 각 메소드의 코드만 따로 놓고 봐서는 알 수 없기 때문에 발견하기 쉽지 않다. 이외에도 여러 스레드가 동시에 여러 뮤텍스를 획득하려는 경우를 포함해서 교착 상태에 빠질 수 있는 조건은 더 있다. Rust의 빌림 시스템은 교착 상태를 막을 수 없다.
최선의 예방책은 임계 영역을 작게 가져가는 것이다. 즉, 임계 영역에 들어가면 필요한 일만 하고 빨리 빠져나와야 한다.

교착 상태

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 스레드가 자신이 이미 주고 있는 락을 획득하려 들면 교착 상태에 빠질 수 있다.
 - 채널을 쓸 때도 교착 상태에 빠질 가능성이 있다. 예를 들어, 두 스레드가 상대방에게서 오는 메시지를 기다리고 있으면 бл록될 수 있다. 하지만 여기서도 프로그램의 설계를 좋게 가져가면 사실상 높은 확률로 그런 일이 벌어지지 않으리라는 확신을 가질 수 있다. 앞서 봤던 역색인 작성기 같은 파이프라인에서는 데이터의 흐름이 비순환적이다. 이런 프로그램은 Unix 셸 파이프라인처럼 작동하기 때문에 교착 상태에 빠질 가능성이 거의 없다.

- Mutex::lock()은 JoinHandle::join()과 같은 이유로, 즉 다른 스레드가 패닉에 빠졌을 때 실패를 매끄럽게 처리하기 위해서 Result를 반환한다.
 - handle.join().unwrap()이라고 쓰면 Rust에게 한 스레드의 패닉을 다른 스레드로 전파하라고 말하는 것이다. mutex.join().unwrap()도 마찬가지다.
- 스레드가 Mutex를 획득한 상태에서 패닉에 빠지면 Rust는 그 Mutex를 오염되었다(Poisoned)고 표시한다.
 - 이 오염된 Mutex에 대고 lock을 호출하면 Result가 Err로 반환된다.
 - 이 뒤에 오는 .unwrap() 호출은 Rust에 그런 일이 벌어지면 패닉에 빠지라고 말하는 것이다.
 - 따라서 다른 스레드의 패닉이 이쪽으로 전파된다.

- 오염된 뮤텍스가 생기는 상황을 어떻게 바라봐야 할까?
 - 오염이란 말이 위험하게 들리긴 해도 이 시나리오가 꼭 치명적인 것만은 아니다. 패닉은 안전하다. 한 스레드가 패닉에 빠지더라도 프로그램의 나머지 부분은 안전한 상태로 유지된다.
 - 패닉에 빠질 때 뮤텍스가 오염되는 이유는 정의되지 않은 동작에 대한 두려움 때문이라기보다는, 불변성(Invariant)을 이용해 프로그래밍해왔을 여러분에 대한 걱정 때문이라고 봐야 한다.
 - 프로그램이 임계 영역에서 패닉에 빠지는 바람에 하던 일을 제대로 마치지 못한 채로 빠져나오면, 보호된 데이터의 필드 일부를 제대로 업데이트하지 못하면서 불변성을 깨뜨릴 가능성이 있다.
 - 따라서 Rust는 뮤텍스를 오염시켜서 다른 스레드가 자신도 모르게 문제의 영역으로 들어와 일을 더 악화시키는 걸 막는다. 그럼에도 불구하고 상호 배제를 완벽히 유지한 채로 오염된 뮤텍스를 잠그고 안에 있는 데이터에 접근하는 것이 가능한데, 이 부분은 문서에 있는 `PoisonError::into_inner()`에 관한 내용을 참고하자.

뮤텍스를 이용한 멀티 컨슈머 채널

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 앞서 우리는 Rust의 채널이 멀티 프로듀서, 싱글 컨슈머라고 언급한 바 있다.
 - 좀 더 구체적으로 말하면 채널에는 Receiver가 하나 뿐이다.
여러 스레드가 `mpsc` 채널 하나를 공유된 작업 목록으로 쓰는 스레드 풀은 있을 수 없다.

뮤텍스를 이용한 멀티 컨슈머 채널

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 앞서 우리는 Rust의 채널이 멀티 프로듀서, 싱글 컨슈머라고 언급한 바 있다.
- 하지만 표준 라이브러리에 있는 기능만으로 만들어 쓸 수 있는 아주 간단한 우회책이 있다.
Receiver를 Mutex로 감싸서 공유해 쓰는 것이다.

```
● ● ●

pub mod shared_channel {
    use std::sync::mpsc::{channel, Receiver, Sender};
    use std::sync::{Arc, Mutex};

    // A thread-safe wrapper around a `Receiver`
    #[derive(Clone)]
    pub struct SharedReceiver<T>(Arc<Mutex<Receiver<T>>);

    impl<T> Iterator for SharedReceiver<T> {
        type Item = T;

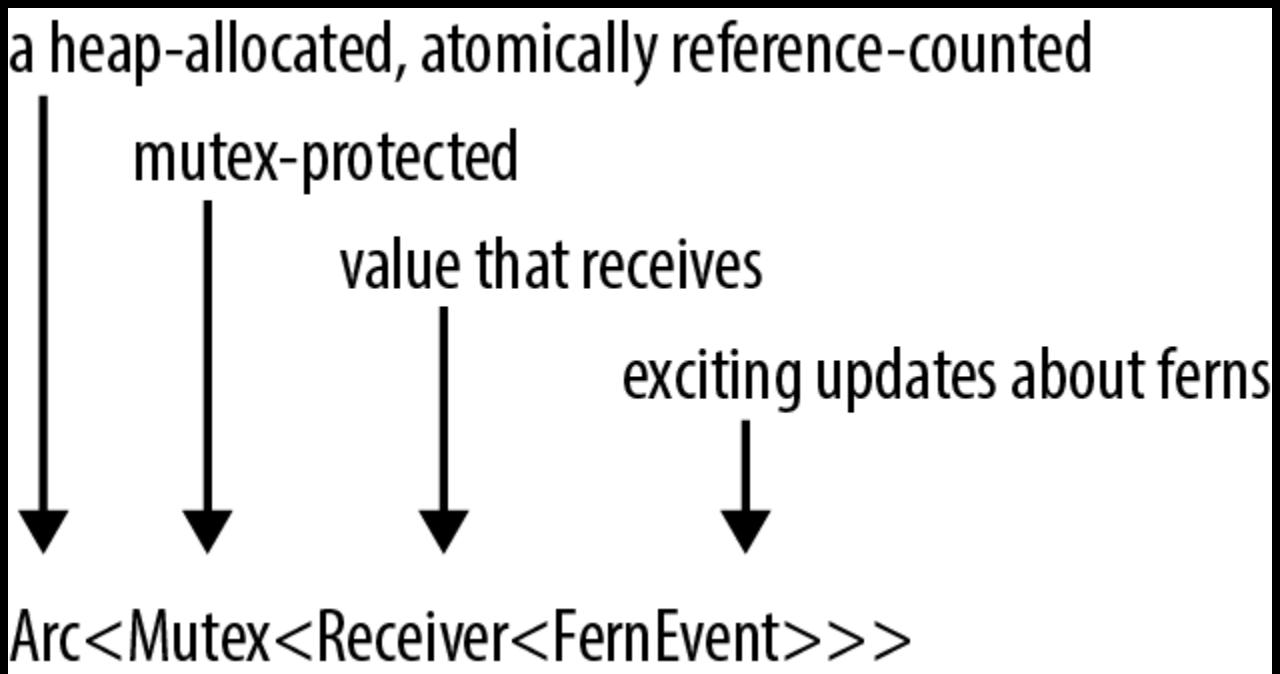
        // Get the next item from the wrapped receiver
        fn next(&mut self) -> Option<T> {
            let guard = self.0.lock().unwrap();
            guard.recv().ok()
        }
    }

    // Create a new channel whose receiver can be shared across threads
    // This returns a sender and a receiver, just like the stdlib's `channel()`,
    // and sometimes works as a drop-in replacement
    pub fn shared_channel<T>() -> (Sender<T>, SharedReceiver<T>) {
        let (sender, receiver) = channel();
        (sender, SharedReceiver(Arc::new(Mutex::new(receiver))))
    }
}
```

뮤텍스를 이용한 멀티 컨슈머 채널

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 앞서 우리는 Rust의 채널이 멀티 프로듀서, 싱글 컨슈머라고 언급한 바 있다.
 - 하지만 표준 라이브러리에 있는 기능만으로 만들어 쓸 수 있는 아주 간단한 우회책이 있다.
Receiver를 Mutex로 감싸서 공유해 쓰는 것이다.



읽기/쓰기 락 (RwLock<T>)

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 서버 프로그램은 한 번 읽어오면 거의 바뀔 일이 없는 구성 정보를 가지고 있을 때가 많다.
 - 대부분의 스레드는 구성을 조회하는 데 그치겠지만 구성이 바뀔 가능성이 열려 있으므로 반드시 락을 써서 보호해야 한다.
 - 예를 들어, 서버에게 디스크에 있는 구성을 다시 읽어오라고 요청하면 이미 읽어 온 구성이 바뀔 가능성이 있다.
 - 뮤텍스는 이런 경우에도 통하지만 불필요한 병목을 만든다. 스레드가 구성을 바꾸지 않고 조회하기만 한다면 굳이 줄을 세워 관리할 이유가 없다. 이럴 때 쓰라고 있는 게 바로 읽기/쓰기 락, 즉 RwLock이다.
- 뮤텍스는 lock 메소드만 가지고 있는 반면, 읽기/쓰기 락은 read와 write 메소드가 있다.
 - RwLock::write는 Mutex::lock과 비슷한 메소드로 보호된 데이터의 배타적 mut 접근을 기다린다.
 - RwLock::read는 mut가 아닌 접근을 제공하는 메소드로 여러 스레드가 동시에 읽어도 안전하기 때문에 기다릴 필요가 거의 없다는 이점이 있다.
 - 뮤텍스를 쓰면 보호된 데이터가 임의의 순간에 가질 수 있는 Reader나 Writer가 (아예 없거나) 하나 뿐이다. 반면 읽기/쓰기 락을 쓰면 일반적인 Rust 레퍼런스처럼 Writer 하나를 갖거나 Reader 여럿을 갖거나 둘 중 하나를 선택할 수 있다.

읽기/쓰기 락 (RwLock<T>)

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 이제 FernEmpireApp에 RwLock으로 보호되는 구성을 위한 구조체를 넣어 보자.

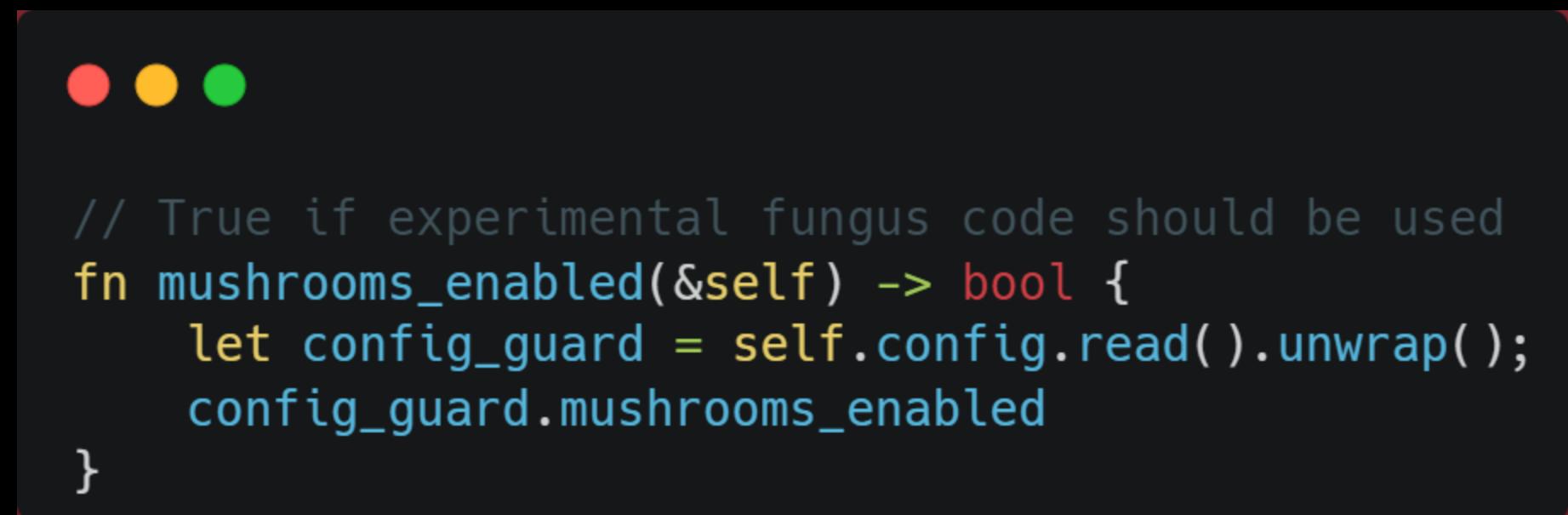
```
use std::sync::RwLock;

struct FernEmpireApp {
    ...
    config: RwLock<AppConfig>,
    ...
}
```

읽기/쓰기 락 (RwLock<T>)

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 이제 FernEmpireApp에 RwLock으로 보호되는 구성을 위한 구조체를 넣어 보자.
- 구성을 읽는 메소드는 RwLock::read()를 쓴다.



```
// True if experimental fungus code should be used
fn mushrooms_enabled(&self) -> bool {
    let config_guard = self.config.read().unwrap();
    config_guard.mushrooms_enabled
}
```

읽기/쓰기 락 (RwLock<T>)

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 이제 FernEmpireApp에 RwLock으로 보호되는 구성을 위한 구조체를 넣어 보자.
- 구성을 다시 읽는 메소드는 RwLock::write()를 쓴다.

```
fn reload_config(&self) -> io::Result<()> {
    let new_config = AppConfig::load()?;
    let mut config_guard = self.config.write().unwrap();
    *config_guard = new_config;
    Ok(())
}
```

조건 변수 (Condvar)

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 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.
- 정확히 원하는 조건을 기다리는 편리한 블로킹 API가 이미 마련되어 있을 때도 있다.
- 이를테면 앞서 예로 든 서버가 종료하는 과정에서는 그냥 `JoinHandle::join`을 쓰면 된다.

조건 변수 (Condvar)

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 그러나 기본 제공 블로킹 API가 없을 때는 **조건 변수(Condition Variable)**를 써서 프로그램 안에 직접 원하는 조건을 만들 수 있다.
 - Rust에서는 `std::sync::Condvar` 타입이 조건 변수를 구현하고 있다.
 - Condvar에는 `.wait()`와 `.notify_all()` 메소드가 있다.
`.wait()`는 다른 스레드가 `.notify_all()`을 호출할 때까지 블록된다.
 - 이외에도 특정 `Mutex`가 보호하는 데이터에 관한 참 또는 거짓 조건을 다루기 위한 것이다 보니, `Mutex`와 `Condvar`가 서로 관련이 있다는 걸 늘 염두에 둘 필요가 있다.

조건 변수 (Condvar)

HSPACE Rust 특강
Rust Basic #14 - 동시성

- 조건 변수를 써본 적이 있는 프로그래머들을 위해서 핵심 코드 두 가지만 짚고 넘어가자.
- 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()를 쓴다.

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

- `std::sync::atomic` 모듈에는 락이 없는 동시성 프로그래밍을 위한 원자적 타입이 포함되어 있다.
 - 일부 추가 기능을 제외하면 기본적으로 표준 C++ 원자성 지원과 동일하다.
 - `AtomicIsize`와 `AtomicUsize`는 싱글 스레드 타입 `isize`와 `usize`에 해당하는 공유된 정수 타입이다.
 - `AtomicI8`, `AtomicI16`, `AtomicI32`, `AtomicI64`를 비롯해서 `AtomicU8`과 같은 이들의 부호 없는 버전은 싱글 스레드 타입 `i8`과 `i16` 등에 해당하는 공유된 정수 타입이다.
 - `AtomicBool`은 공유된 `bool` 값이다.
 - `AtomicPtr<T>`는 안전하지 않은 포인터 타입 `*mut T`의 공유된 값이다.

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

```
● ● ●  
  
use std::sync::atomic::{AtomicIsize, Ordering};  
  
let atom = AtomicIsize::new(0);  
atom.fetch_add(1, Ordering::SeqCst);
```

- 이 메소드들은 특화된 기계어 명령으로 컴파일된다. x86-64 아키텍처에서 .fetch_add() 호출은 lock incq 명령으로 컴파일되는 반면, n += 1은 평범한 incq 명령이나 같은 일을 하는 다른 변형으로 컴파일된다. 또 일반적인 로드나 스토어와 달리 이들은 합법적으로 다른 스레드와 그때그때 영향을 주고받을 수 있기 때문에 Rust 컴파일러는 원자적 연산을 둘러싼 최적화 일부를 포기해야 한다.

- 인수 Ordering::SeqCst는 **메모리 순서(Memory Order)**를 지정한다.
 - 메모리 순서는 데이터베이스의 트랜잭션 격리 수준과 비슷하다.
이런 것들은 성능과는 별개로 결과보다 원인을 중시하고 반복 없는 시간을 추구하는 등의 철학적 개념에 대해서 얼마나 신경 쓰고 있는지를 시스템에게 말해 준다.
 - 메모리 순서는 프로그램의 정확성을 결정짓는 중요한 요소로 이해하기 어렵고 추론하기 까다롭다.
다행인 건 SQL 데이터베이스를 SERIALIZABLE 모드로 쓸 때 벌어지는 성능 저하와 달리,
가장 엄격한 메모리 순서인 순차적 일관성을 고르더라도 성능 저하가 적을 때가 많다는 점이다.
따라서 멀 써야 좋을지 모를 때는 그냥 Ordering::SeqCst를 쓰면 된다.
 - Rust는 표준 C++ 동시성 지원이 가진 다른 여러 메모리 순서를 물려받았다.
자세한 내용은 https://en.cppreference.com/w/cpp/atomic/memory_order을 참고하자.

- 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 좋을지 모른다는 건데, 이럴 때 공유된 `AtomicBool`을 쓰면 간단히 해결할 수 있다.

```
● ● ●

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

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

- 위 코드는 초기값이 `false`인 `AtomicBool`을 힙에 할당하고, 이를 가리키는 두 개의 `Arc<AtomicBool>` 스마트 포인터를 만든다. `cancel_flag`라고 된 첫 번째 스마트 포인터는 메인 스레드에 머물고, `worker_cancel_flag`라고 된 두 번째 스마트 포인터는 워커 스레드로 이동한다.

- 워커의 스레드는 다음과 같다.

```
● ● ●

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)
});
```

- 메인 스레드가 워커 스레드를 취소하기로 결정하면 AtomicBool에 true를 저장하고 스레드가 종료될 때까지 기다린다.

```
● ● ●  
  
// Cancel rendering  
cancel_flag.store(true, Ordering::SeqCst);  
  
// Discard the result, which is probably `None`  
worker_handle.join().unwrap();
```

- 네트워킹 코드를 작성 중이라고 하자.
여기에 패킷을 처리할 때마다 값이 증가하는 카운터를 전역 변수로 두려고 한다.
- 아래 코드는 잘 컴파일되지만 한 가지 문제가 있다.
`PACKETS_SERVED`가 변경할 수 있게 선언되어 있지 않아서 값을 증가시킬 수 없다는 거다.

```
// Number of packets the server has successfully handled
static PACKETS_SERVED: usize = 0;
```

- 네트워킹 코드를 작성 중이라고 하자.

여기에 패킷을 처리할 때마다 값이 증가하는 카운터를 전역 변수로 두려고 한다.

- Rust는 변경할 수 있는 전역 상태를 막기 위해서라면 합리적인 선에서 수단과 방법을 가리지 않는다. 물론, `const`로 선언하는 상수는 변경할 수 없다. 정적 변수 역시 기본적으로는 변경할 수 없으므로 이를 참조하는 `mut` 레퍼런스를 만들 수 있는 방법이 없다. `static`은 `mut`로 선언할 수 있지만, 이를 접근하는 건 안전하지 않다. 이러한 규칙이 생겨난 배경에는 스레드 안전성에 관한 Rust의 고집이 자리하고 있다.
- 또 변경할 수 있는 전역 상태는 소프트웨어 엔지니어링에 불행한 결과를 초래하는데, 프로그램의 다양한 부분이 점점 더 단단하게 결합되는 경향이 생기면서 테스트하기도 어렵고, 원가를 바꾸기도 까다롭게 변한다. 그럼에도 불구하고 다른 합리적인 대안이 없을 때도 있어서 변경할 수 없는 정적 변수를 선언하는 안전한 방법을 찾는 게 더 낫다.

- `PACKETS_SERVED`의 값을 증가시킬 수 있게 지원하면서도 스레드 안전성을 잃지 않는 가장 단순한 방법은 이를 원자적 정수로 만드는 것이다.

```
● ● ●  
use std::sync::atomic::AtomicUsize;  
  
static PACKETS_SERVED: AtomicUsize = AtomicUsize::new(0);
```

- 이런 식으로 전역 변수를 선언하고 나면 패킷 카운트를 올리는 건 간단하다.

```
● ● ●  
use std::sync::atomic::Ordering;  
  
PACKETS_SERVED.fetch_add(1, Ordering::SeqCst);
```

- 원자적 전역 변수는 단순한 정수와 `bool`로 제한된다.
이외의 다른 타입으로 된 전역 변수를 만들면 두 가지 문제를 풀어야 한다.
- 첫 번째로 변수가 어떻게든 스레드 안전성을 가져야 한다. 그렇지 않으면 전역 변수가 될 수 없다.
정적 변수가 안전성을 가지려면 `Sync`이면서 `mut`가 아니어야 한다. 다행히도 이 문제의 해결책은 이미 알고 있다.
Rust에는 바뀌는 값을 안전하게 공유하기 위한 타입이 있는데, `Mutex`, `RwLock`, 원자적 타입이 바로 그것이다.
이들 타입은 `mut`가 아닌 타입으로 선언해도 수정할 수 있는데, 이게 바로 그들의 일이다.

- 원자적 전역 변수는 단순한 정수와 `bool`로 제한된다.
이외의 다른 타입으로 된 전역 변수를 만들면 두 가지 문제를 풀어야 한다.
 - 두 번째로 `static`을 초기화하는 코드는 특별히 `const`로 표시된 함수만 호출할 수 있다.
이런 함수는 컴파일러가 컴파일 과정에서 평가할 수 있기 때문에 결과가 결정성을 띠고
다른 상태나 I/O가 아니라 주어진 인수에만 의존한다는 특성이 있다.
따라서 컴파일러가 이런 식으로 해당 계산의 결과를 컴파일 시점 상수로 끄워 넣을 수 있다.
`C++`의 `constexpr`이 비슷한 기능을 한다.
 - `Atomic` 타입의 생성자는 모두 `const` 함수다. 앞에서 `AtomicUsize`를 `static`으로 만들 수 있었던 건
바로 이런 이유 때문이다. `String`, `Ipv4Addr`, `Ipv6Addr` 같은 일부 타입에도 간단한 `const` 생성자가 있다.

- 나만의 `const` 함수를 정의할 수도 있는데, 함수의 시그니처 앞에 `const`을 붙이면 된다.
 - Rust는 `const` 함수가 할 수 있는 작업의 규모를 작게 제한해 두고 있는데, 이들 작업은 결과가 비결정성을 띠지 않으면서도 충분히 유용하다.
 - `const` 함수는 제네릭 인수로 수명만 받을 수 있고, 타입은 받을 수 없으며, `unsafe` 블록 안이라고 해도 메모리를 할당하거나 원시 포인터를 쓰는 작업은 할 수 없다.
 - 하지만 산술 연산, Short-circuit Evaluation이 없는 논리 연산, 다른 `const` 함수를 쓸 순 있다.

- 나만의 `const` 함수를 정의할 수도 있는데, 함수의 시그니처 앞에 `const`을 붙이면 된다.
 - 예를 들어, 다음 예제처럼 `static`과 `const`을 손쉽게 정의하게 해서 코드 중복을 줄이는 편리한 함수를 만들 수 있다.



```
● ● ●

const fn mono_to_rgba(level: u8) -> Color {
    Color {
        red: level,
        green: level,
        blue: level,
        alpha: 0xFF,
    }
}

const WHITE: Color = mono_to_rgba(255);
const BLACK: Color = mono_to_rgba(000);
```

- 이런 기법을 엮어서 다음과 같은 코드를 작성하고 싶을 수 있다.

```
// Error: Calls in statics are limited to constant functions,  
//        tuple structs, and tuple variants  
static HOSTNAME: Mutex<String> = Mutex::new(String::new());
```

- 안타깝게도 `AtomicUsize::new()`와 `String::new()`는 `const fn`이지만 `Mutex::new()`는 아니다. 이런 제한을 피해 가려면 `lazy_static` 크레이트를 써야 한다.

- `lazy_static!`을 쓰면 정적 데이터에 접근할 때마다 약간의 성능 비용이 발생한다.
 - 왜냐하면 해당 구현이 일회성 초기화를 위해 설계된 저수준 동기화 기본 요소인 `std::sync::Once`를 쓰기 때문이다.
 - 접근할 때마다 프로그램은 이면에서 초기화가 이미 진행됐는지 확인하기 위해 원자적 로드 명령을 실행한다.

```
● ● ●

use lazy_static::lazy_static;
use std::sync::Mutex;

lazy_static! {
    static ref HOSTNAME: Mutex<String> = Mutex::new(String::new());
}
```

감사합니다.

utilForever@gmail.com

<https://github.com/utilForever>

Facebook, Twitter: @utilForever