

Konkuk University GDSC + EDGE 스터디

Week 8: Concurrency, Part 1

Chris Ohk

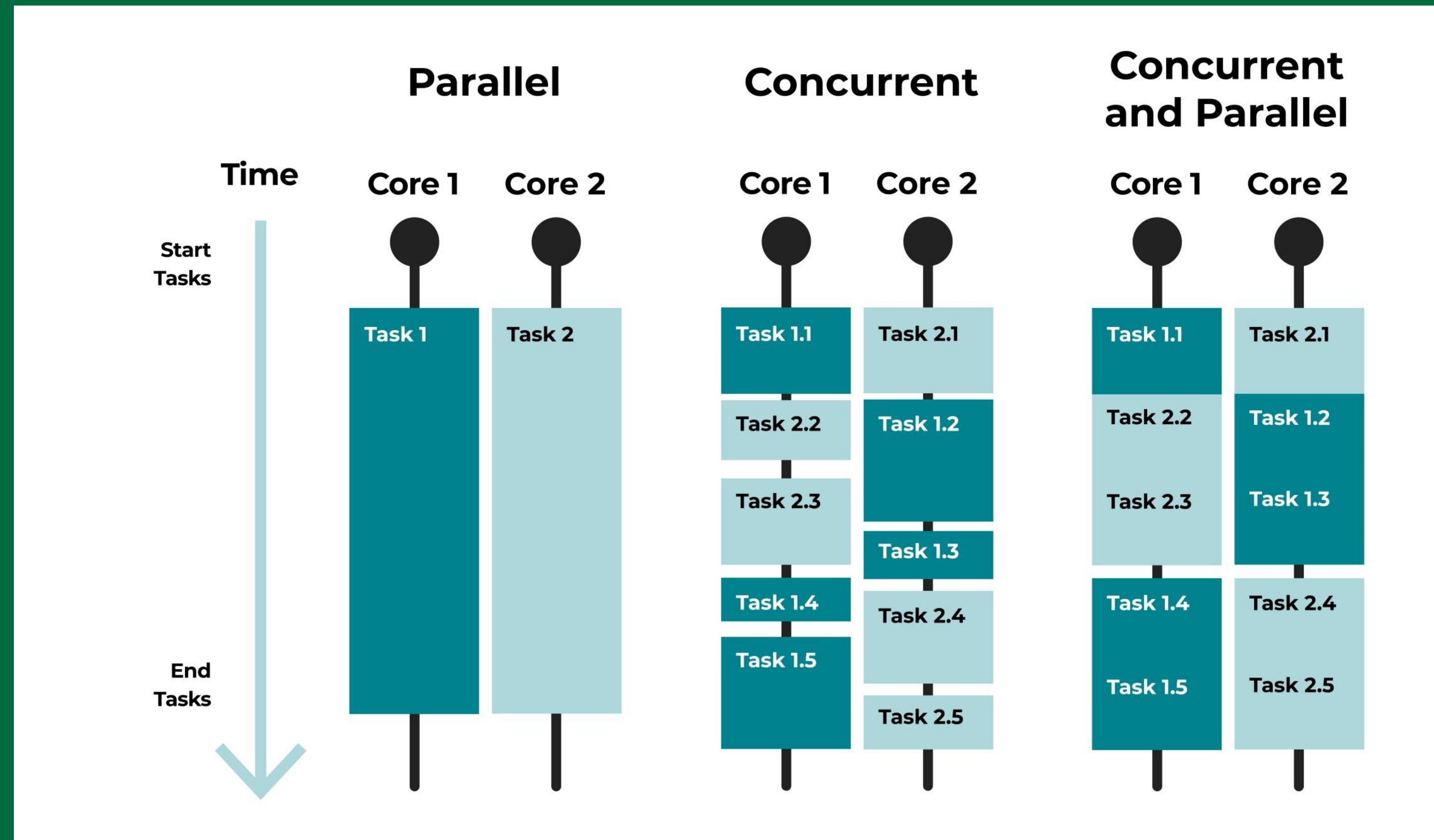
utilForever@gmail.com

목차

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- Fork-Join 병렬 처리
- 채널

- Concurrency vs Parallelism



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

Fork-Join 병렬 처리

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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);

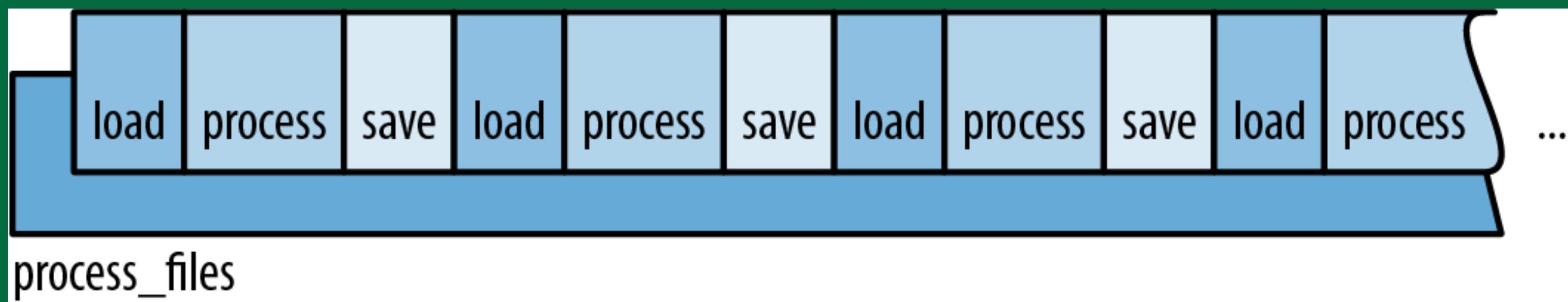
        save(&document, results)?;
    }

    Ok(())
}
```

Fork-Join 병렬 처리

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

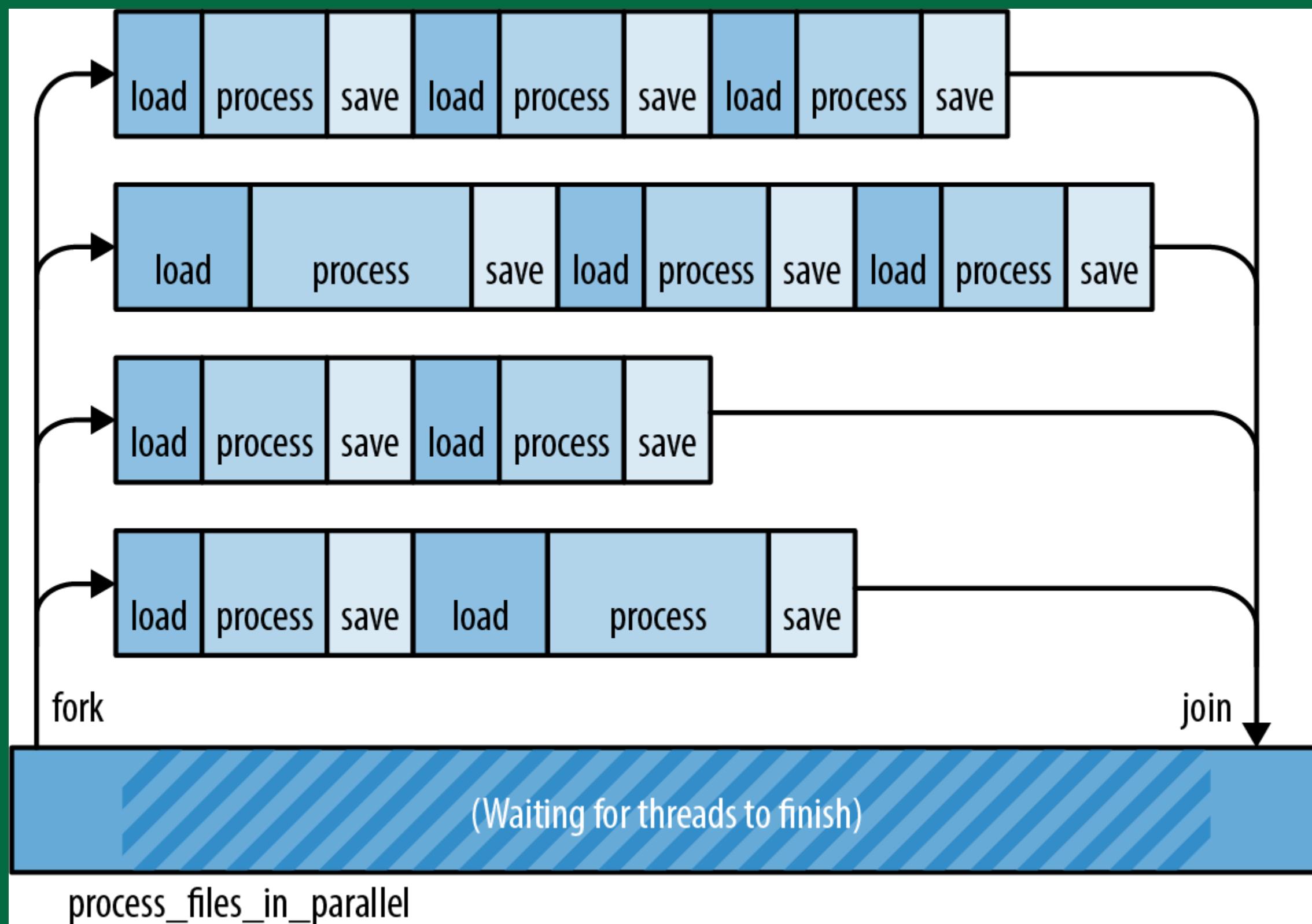
- 이 프로그램은 다음 그림과 같이 실행된다.



Fork-Join 병렬 처리

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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 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 유지하는데 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

- `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");  
});
```

spawn과 join

- 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(())
}
```

- 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.

- 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 `? 연산자`를 써서 자식 스레드의 I/O 오류를 부모 스레드에게 명시적으로 전파한다.

- 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

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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)));
    }

    // ...
}
```

- 왜 컴파일 오류가 발생할까?
 - Rust가 문제 삼는 부분은 `spawn`에 넘기고 있는 클로저의 수명으로, 컴파일러가 '도움이 될까 싶어' 제시한 앞의 메시지는 사실 아무런 도움이 되지 않는다.
 - `spawn`은 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_parallel`이 다른 스레드가 끝나기도 전에 서둘러 떠날 가능성이 있다. 이렇게 되면 자식 스레드는 결국 메인 스레드가 이미 해제한 `glossary`를 사용하려고 하는 상황에 놓일 수 있다. 그러다 엉친 데 덮친 격으로 메인 스레드가 `glossary`를 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

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 어떻게 해결할 수 있을까?
 - 스레드 간 레퍼런스 공유를 지원하기엔 spawn이 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 glossary를 통으로 clone하는 게 그나마 안전한 대안이지만, 덩치가 너무 커서 웬만하면 피하고 싶다.
다행히도 표준 라이브러리는 원자적인 레퍼런스 카운팅이라는 또 다른 대안을 제공한다. 바로 Arc다!

변경할 수 없는 데이터 공유하기

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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)
        }));
    }

    // ...
}
```

- 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.

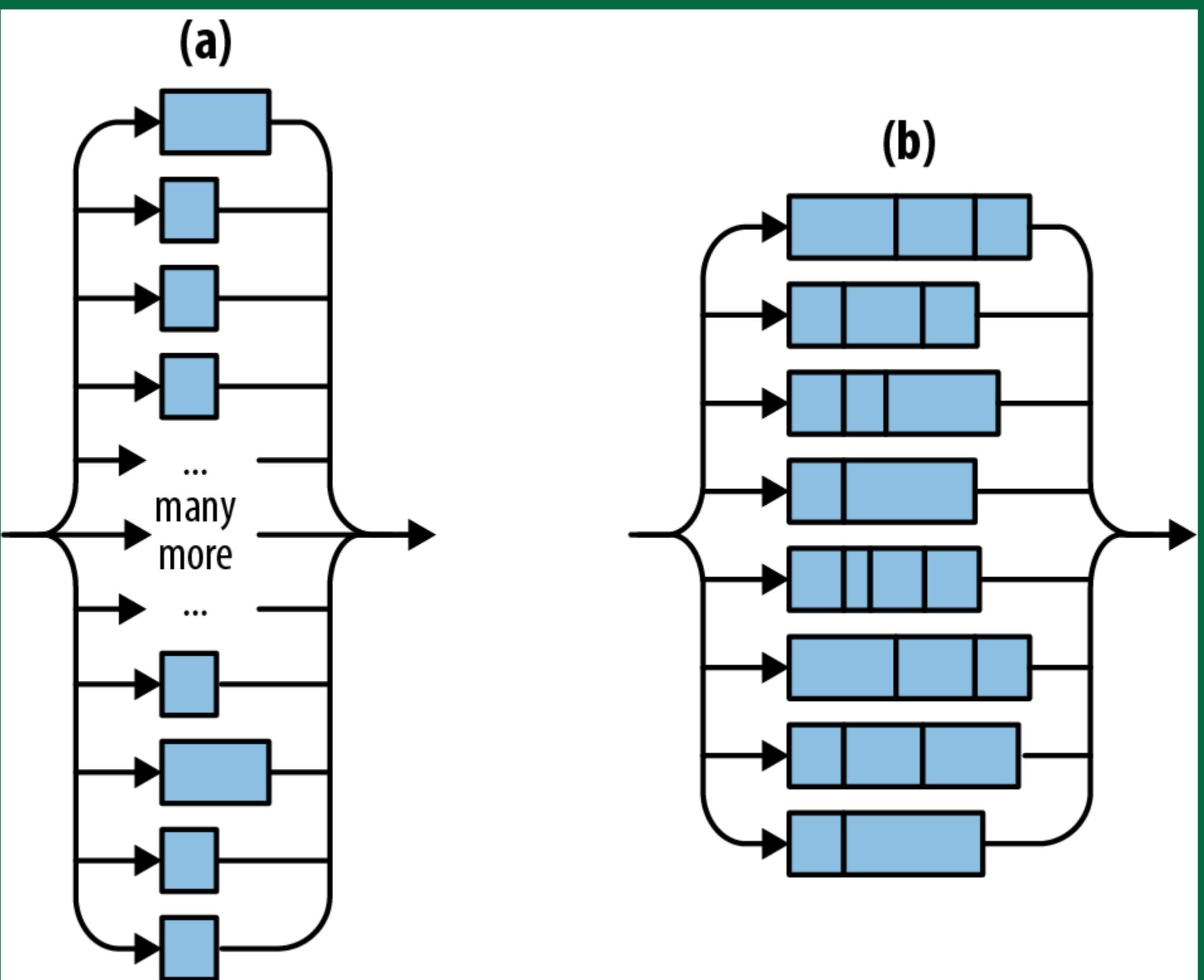
```
● ● ●

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);
});
```

- 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.



- 다음은 Rayon을 사용한 `process_files_in_parallel` 함수의 구현 코드다.

```
● ● ●

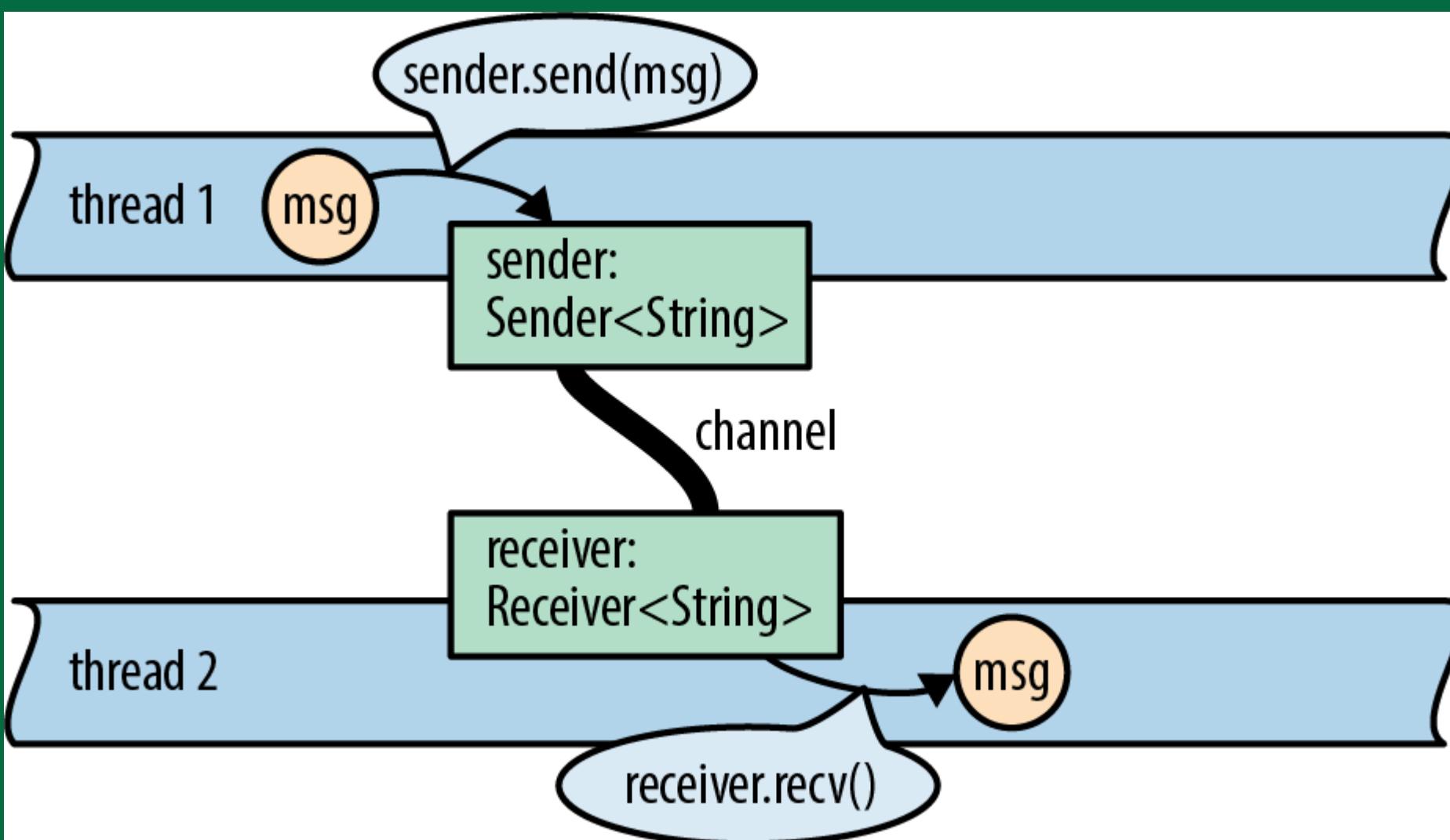
use rayon::prelude::*;

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(()))
}
```

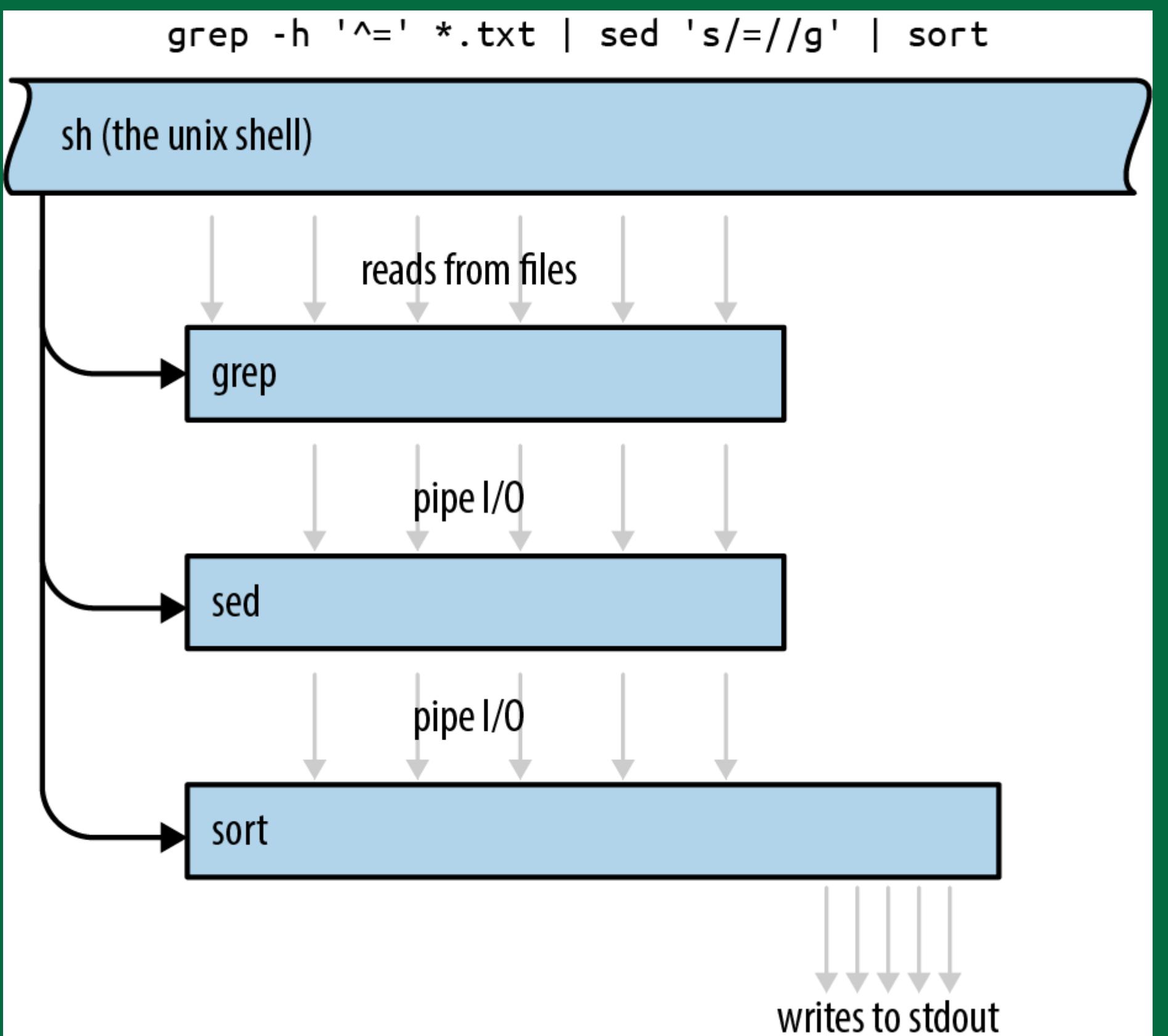
- Rayon의 특징
 - Rayon은 이면에서 작업 훔치기(Work-Stealing)이라고 하는 기법을 써서 스레드 간의 작업량을 동적으로 조절한다.
이 기법은 앞서 나온 spawn/join을 사용해 직접 작업을 미리 분할할 때보다 전체 CPU 사용율을 더 높게 가져갈 수 있다.
 - 또한 Rayon은 스레드 간 레퍼런스 공유를 지원한다.
이면에서 벌어지는 모든 병렬 처리는 reduce_with가 복귀하기 전에 반드시 끝난다. 클로저가 여러 스레드에서
호출되는 상황임에도 불구하고 glossary를 process_file에 넘길 수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



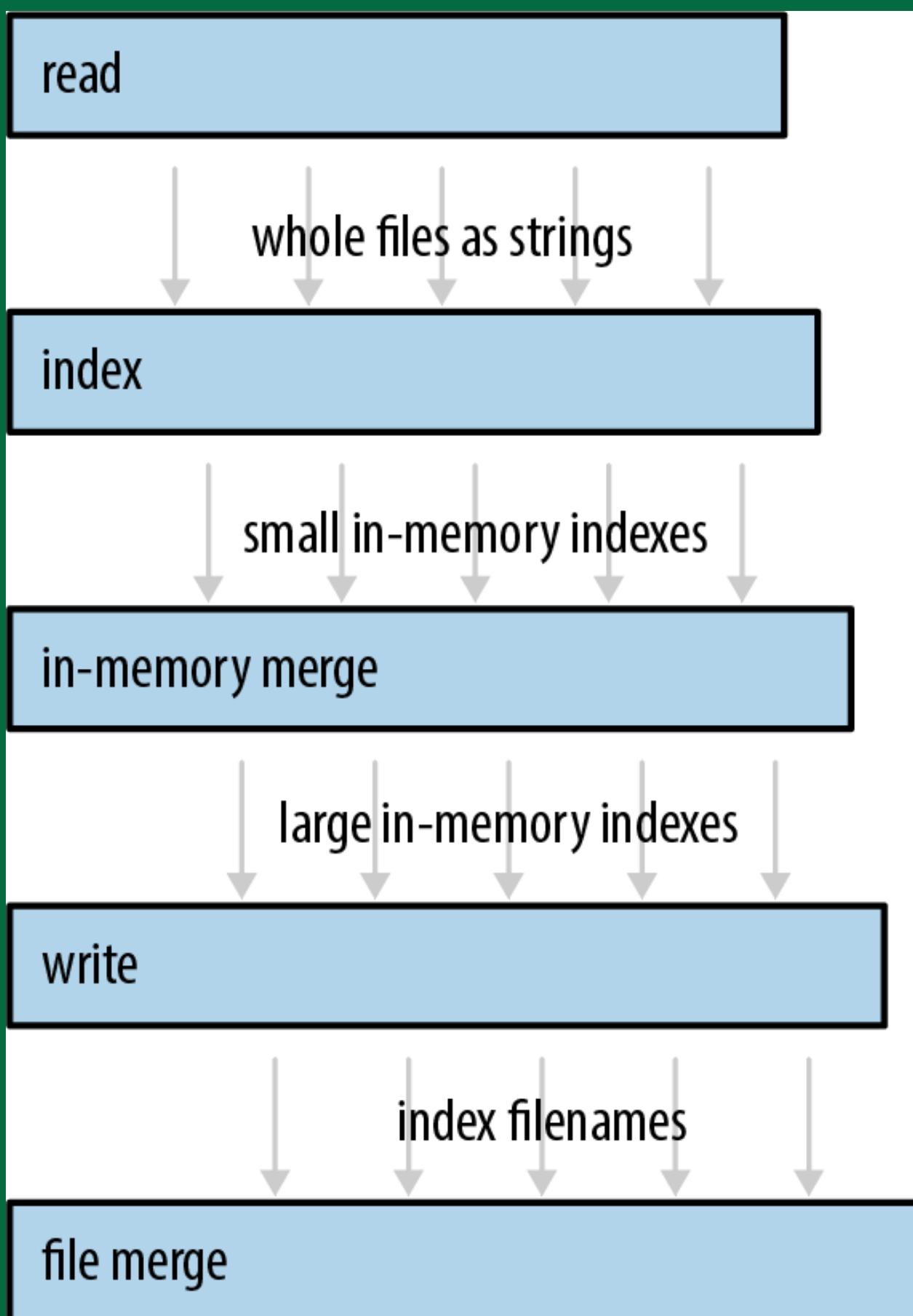
- **채널(Channel)**은 한 스레드에서 다른 스레드로 값을 보내기 위한 단방향 도관이다.
 - 채널을 쓰면 스레드가 서로 값을 주고 받으며 통신할 수 있다.
잠금이나 공유된 메모리를 쓰지 않고도 스레드가 서로 협업할 수 있게 해주는 아주 단순한 장치다.
 - 사실 채널은 새로운 기술이 아니다.
 - 얼랭(Erlang)은 30년 전부터 격리된 프로세스와 메시지 전달을 사용해 왔다.
 - Unix의 파이프가 사용된 지는 거의 50년이 다 되어 간다.



- 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.

- 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(())
});
```

- 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.

```
● ● ●

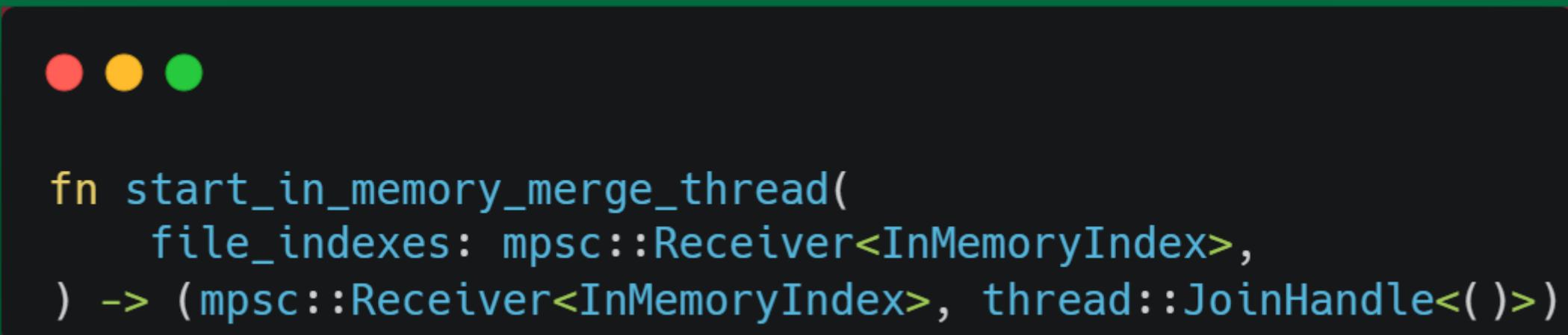
fn start_file_indexing_thread(
    texts: mpsc::Receiver<String>,
) -> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>) {
    let (sender, receiver) = mpsc::channel();

    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

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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
}
```

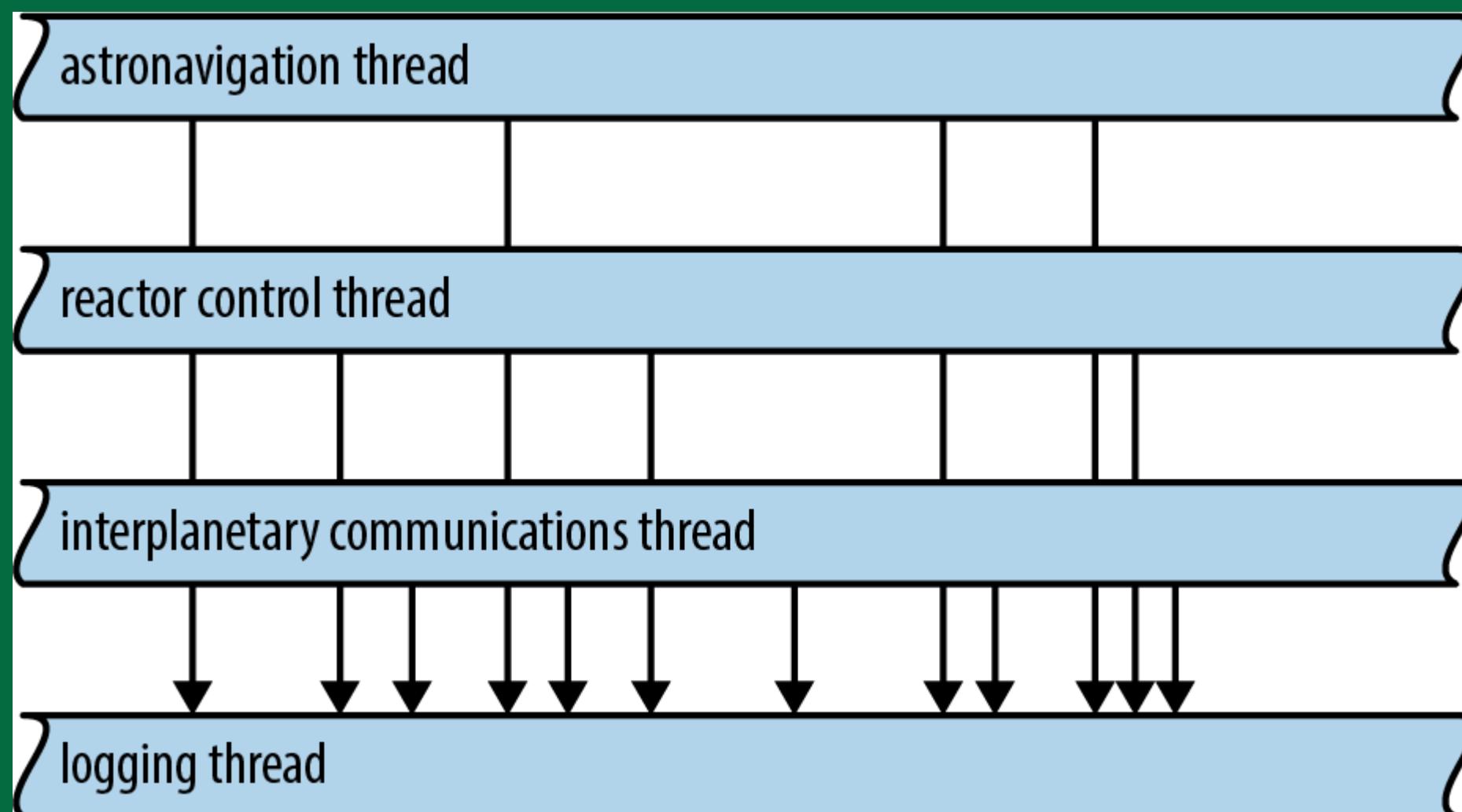
채널의 기능과 성능

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- `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

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- `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.

- `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 `Mutex`가 필요하다. 이는 뒤에서 살펴볼 예정이다.

채널의 기능과 성능

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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.

- 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.

- 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

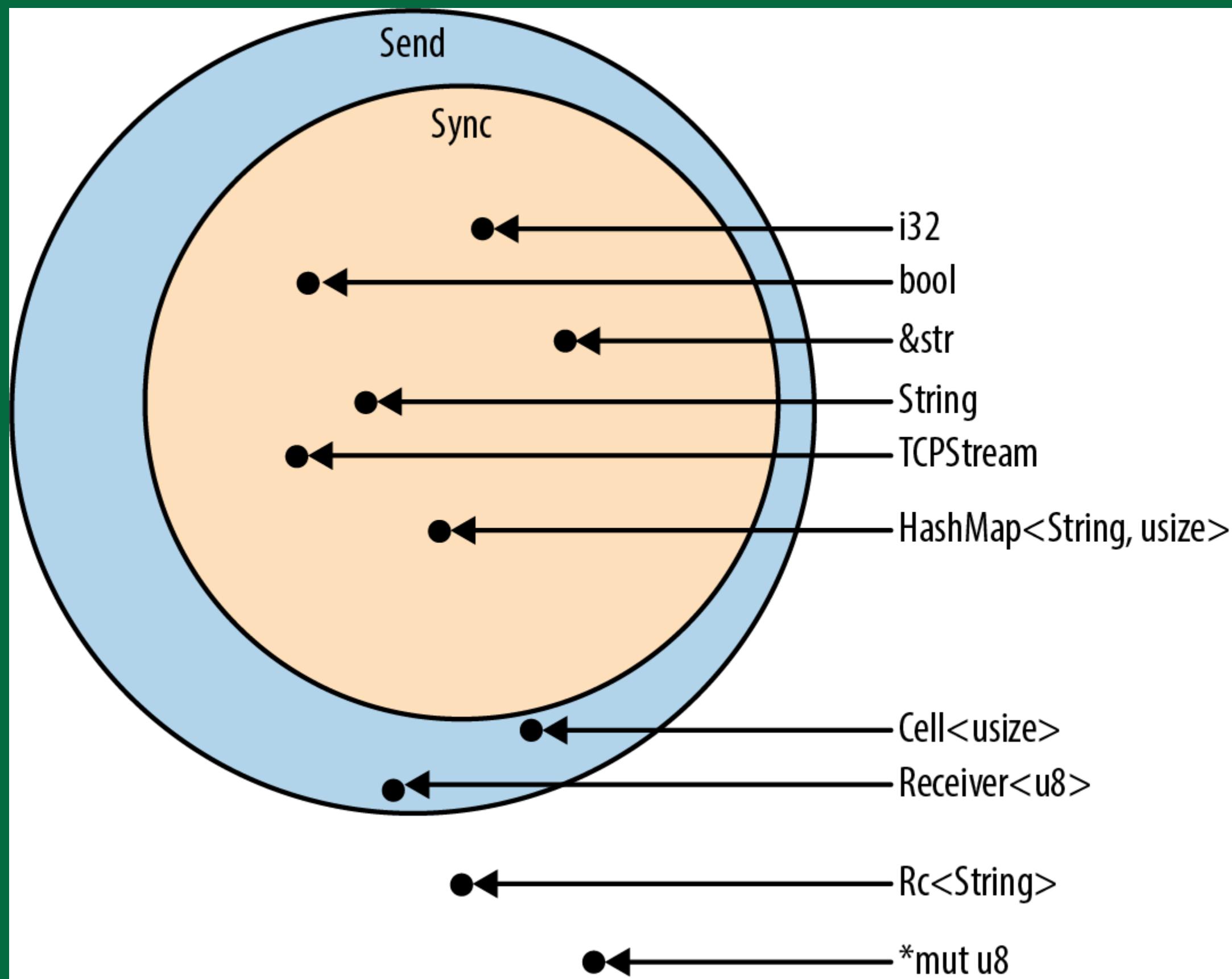
Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

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



스레드 안전성 : Send와 Sync

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

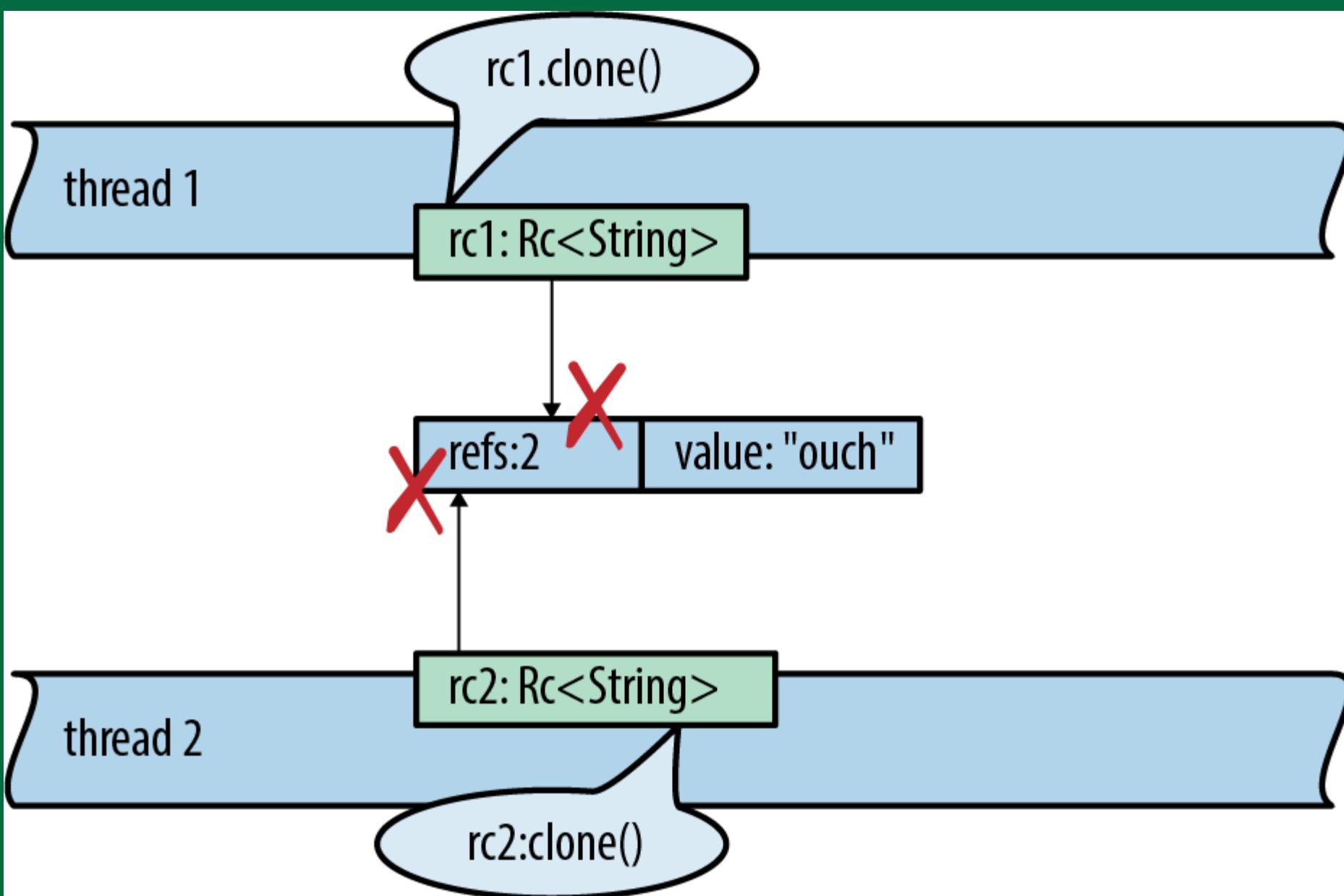
- 예외 사항
 - Send지만 Sync가 아닌 타입도 있는데, 이는 보통 의도적인 결정일 때가 많다.
`mpsc::Receiver`는 이런 특성을 이용해서 `mpsc` 채널의 받는 쪽을 한 번에 한 스레드에서만 쓸 수 있게 보장한다.
 - 소수지만 Send도 아니고 Sync도 아닌 타입은 대부분 스레드 세이프하지 않은 방식으로 가변성을 이용한다.
예를 들어, 레퍼런스 카운트를 쓰는 스마트 포인터 타입 `std::rc::Rc<T>`를 보자.

스레드 안전성 : Send와 Sync

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 예외 사항

- `Rc<String>`이 Sync여서 `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

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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

Konkuk University GDSC + EDGE 스터디
Week 8: Concurrency, Part 1

- 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()
    }
}
```

- <https://www.rust-lang.org/>
- <https://doc.rust-lang.org/book/>
- <https://tourofrust.com/>
- The Rust Programming Language (No Starch Press, 2019)
- Programming Rust, 2nd Edition (O'Reilly, 2021)

Thank you!