세상에서 제일 쉬운 러스트 프로그래밍

12장. 멀티스레딩

윤인도

freedomzero91@gmail.com

멀티스레딩(multithreading)

하나의 프로세서에서 여러 개의 스레드를 만들어 작업을 "동시에" 처리하는 방법

- 병렬적 : 여러 개의 작업이 동시에 실행되는 것
- 동시적 : 여러 개의 작업이 번갈아가며 실행되는 것

스레드 스폰

프로세스에서 스레드를 만드는 행위

싱글 스레드 스폰하기

- 모든 프로그램은 메인 스레드로부터 시작
- 별도의 스레드를 만들지 않은 경우, 프로그램은 메인 스레드에서 실행
- 메인 스레드에 새로운 스레드를 추가하는 것이 스레드 스폰

```python
from threading import Thread

from time import sleep


def func1():
    for i in range(0, 10):
        print(f"Hello, can you hear me?: {i}")
        sleep(0.001)

thread = Thread(target=func1)
thread.start()

for i in range(0, 5):
    print(f"Hello from the main thread: {i}")
    sleep(0.001)
```

실행 결과

메인 스레드와 생성된 스레드가 작업을 번갈아 수행

```
Hello, can you hear me?: 0
Hello from the main thread: 0
Hello, can you hear me?: 1
Hello from the main thread: 1
Hello, can you hear me?: 2
Hello from the main thread: 2
Hello, can you hear me?: 3
Hello from the main thread: 3
Hello, can you hear me?: 4
Hello from the main thread: 4
Hello, can you hear me?: 5
Hello, can you hear me?: 6
Hello, can you hear me?: 7
Hello, can you hear me?: 8
Hello, can you hear me?: 9
```

러스트에서는 `std::thread::spawn` 사용

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

fn main() {
    let handle = thread::spawn(|| {
        for i in 0..10 {
            println!("Hello, can you hear me?: {}", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 0..5 {
        println!("Hello from the main thread: {}", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap(); // 잠시 후에 설명!
}
```

실행 결과

```
Hello from the main thread: 0
Hello, can you hear me?: 0
Hello from the main thread: 1
Hello, can you hear me?: 1
Hello from the main thread: 2
Hello, can you hear me?: 2
Hello, can you hear me?: 3
Hello from the main thread: 3
Hello, can you hear me?: 4
Hello from the main thread: 4
Hello, can you hear me?: 5
Hello, can you hear me?: 6
Hello, can you hear me?: 7
Hello, can you hear me?: 8
Hello, can you hear me?: 9
```

대몬(daemon) 스레드 만들기

- 대몬 스레드는 백그라운드에서 실행되며, 메인 스레드가 종료되면 함께 종료되는 스레드
- 파이썬에서는 `daemon=True`로 설정해서 대몬 스레드를 생성

```python
from threading import Thread
from time import sleep

def func1():
    for i in range(0, 10):
        print(f"Hello, can you hear me?: {i}")
        sleep(0.001)

thread = Thread(target=func1, daemon=True)
thread.start()

for i in range(0, 5):
    print(f"Hello from the main thread: {i}")
    sleep(0.001)
```

실행 결과

```
Hello, can you hear me?: 0
Hello from the main thread: 0
Hello, can you hear me?: 1
Hello from the main thread: 1
Hello from the main thread: 2
Hello, can you hear me?: 2
Hello, can you hear me?: 3
Hello from the main thread: 3
Hello from the main thread: 4
Hello, can you hear me?: 4
```

스폰된 스레드의 작업이 끝나기 전에 메인 스레드의 작업이 끝나고 즉시 프로그램이 종료

러스트에서 `handle.join` 을 호출하지 않으면 스폰된 스레드가 실행 중이더라도 메인 스레드가 종료되고 프로그램이 종료

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

fn main() {
    thread::spawn(|| {
        for i in 0..10 {
            println!("Hello, can you hear me?: {}", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 0..5 {
        println!("Hello from the main thread: {}", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

실행 결과

```
Hello from the main thread: 0
Hello, can you hear me?: 0
Hello from the main thread: 1
Hello, can you hear me?: 1
Hello from the main thread: 2
Hello, can you hear me?: 2
Hello from the main thread: 3
Hello, can you hear me?: 3
Hello from the main thread: 4
Hello, can you hear me?: 4
Hello, can you hear me?: 5
```

`join` 을 사용해 스레드 기다리기

`join` 함수를 사용해 스레드 작업이 끝날 때까지 기다릴 수 있습니다.

```
from threading import Thread
from time import sleep

def func1():
    for i in range(0, 10):
        print(f"Hello, can you hear me?: {i}")
        sleep(0.001)

thread = Thread(target=func1)
thread.start()
thread.join()

for i in range(0, 5):
    print(f"Hello from the main thread: {i}")
    sleep(0.001)
```

실행 결과

```
Hello, can you hear me?: 0
Hello, can you hear me?: 1
Hello, can you hear me?: 2
Hello, can you hear me?: 3
Hello, can you hear me?: 4
Hello, can you hear me?: 5
Hello, can you hear me?: 6
Hello, can you hear me?: 7
Hello, can you hear me?: 8
Hello, can you hear me?: 9
Hello from the main thread: 0
Hello from the main thread: 1
Hello from the main thread: 2
Hello from the main thread: 3
Hello from the main thread: 4
```

러스트에서는 `join()`을 사용해 스폰된 스레드가 끝까지 실행되기를 기다렸다가 메인 스레드를 실행

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

fn main() {
    let handle = thread::spawn(|| {
        for i in 0..10 {
            println!("Hello, can you hear me?: {}", i);
            thread::sleep(Duration::from_millis(1));
        }
    });
    handle.join().unwrap();

    for i in 0..5 {
        println!("Hello from the main thread: {}", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

실행 결과

```
Hello, can you hear me?: 0
Hello, can you hear me?: 1
Hello, can you hear me?: 2
Hello, can you hear me?: 3
Hello, can you hear me?: 4
Hello, can you hear me?: 5
Hello, can you hear me?: 6
Hello, can you hear me?: 7
Hello, can you hear me?: 8
Hello, can you hear me?: 9
Hello from the main thread: 0
Hello from the main thread: 1
Hello from the main thread: 2
Hello from the main thread: 3
Hello from the main thread: 4
```

GIL(Global interpreter lock)

- GIL은 한 번에 하나의 스레드만 파이썬 바이트코드를 실행하도록 하기 위해 인터프리터에서 사용되는 락(lock)
- 스레드가 GIL을 획득하면 파이썬 바이트코드를 실행할 수 있지만 다른 모든 스레드는 GIL이 해제될 때까지 파이썬 코드를 실행할 수 없도록 차단

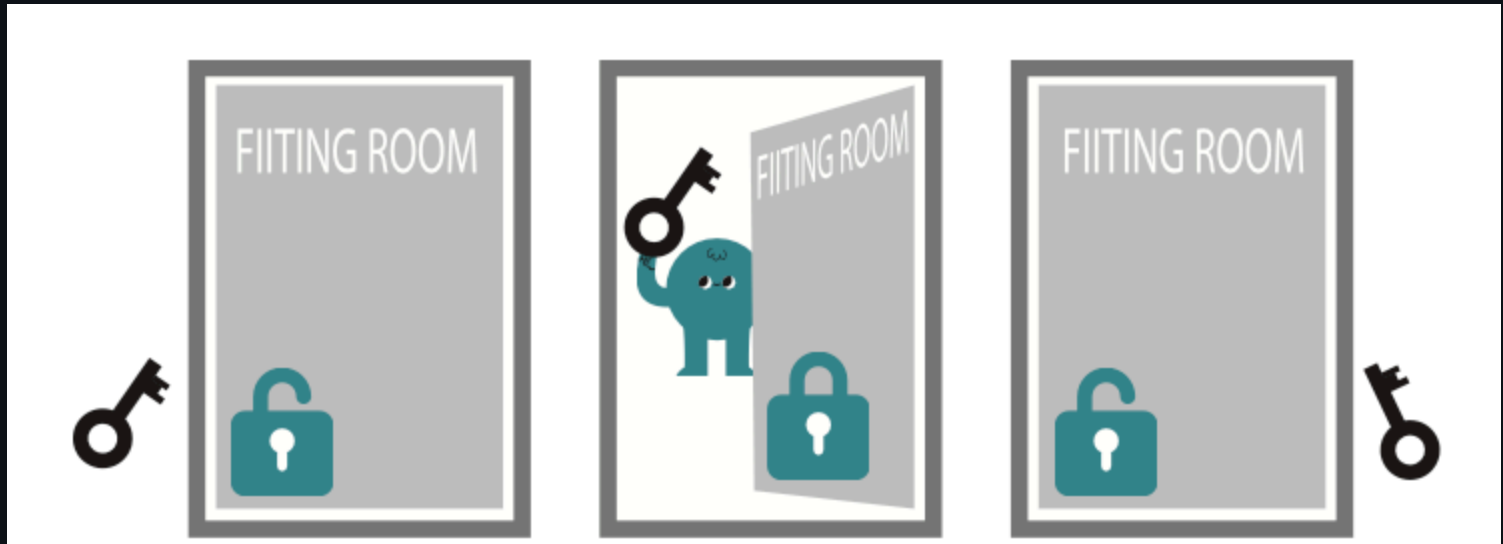

그림 12-1 GIL 작동 원리

GIL의 단점

- GIL은 멀티스레드 프로그램, 특히 CPU를 많이 사용하는 프로그램의 성능을 저해
- 한 스레드가 CPU에 바인딩된 작업을 실행하는 경우, 다른 스레드가 I/O 또는 기타 CPU에 바인딩되지 않은 작업을 수행하려고 해도 현재 스레드의 작업이 끝날 때까지 기다려야 함

GIL 경합(contention) 문제

매우 큰 수를 반복하는 코드

```
import time
import threading

N = 10000000

def count(n):
    for i in range(n):
        pass
```

순차 실행 VS 스레드 실행

```
start = time.time()
count(N)
count(N)
print(f"Elapsed time(sequential): {(time.time() - start) * 1000:.3f}ms")

start = time.time()
t1 = threading.Thread(target=count, args=(N,))
t2 = threading.Thread(target=count, args=(N,))

t1.start()
t2.start()

t1.join()
t2.join()

print(f"Elapsed time(threaded): {(time.time() - start) * 1000:.3f}ms")
```

실행 결과

```
Elapsed time(sequential): 0.4786410331726074
Elapsed time(threaded): 0.4163088798522949
```

스레드를 사용하더라도 `count` 함수를 연속으로 호출한 경우와 실행 속도 차이가 크지 않음

GIL의 한계를 극복하기 위해 파이썬은 몇 가지 기능을 지원합니다.

- 비동기 프로그래밍
- 멀티스레딩
- 멀티프로세싱
- 제한적 GIL 해제

실제 개발자 입장에서 가장 널리 쓰이는 방식은 비동기 프로그래밍과 멀티스레딩

스레드와 소유권

- `std::thread::spawn`에는 흔히 클로저(closure)를 전달함
- 클로저에 `move`를 사용하면 특정 값을 스레드 안으로 이동시킬 수 있음

```
let numbers = vec![1, 2, 3];
thread::spawn(move || {
    for n in numbers {
        println!("{n}");
    }
})
.join()
.unwrap();
```

`move`를 사용하지 않으면 클로저는 `numbers`를 레퍼런스로 사용

```
fn main() {
    let number = 1;
    let handle = thread::spawn(|| {
        println!("{}", number); // `numbers`의 소유권이 스레드로 이동해버림
    });
    handle.join().unwrap();
}
```

실행 결과

```
error[E0373]: closure may outlive the current function, but it borrows `number`,
 which is owned by the current function
 --> src/main.rs:4:32
  |
4 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `number`
5 |         println!("{}", number);
  |                        ------ `number` is borrowed here
```

클로저의 리턴값은 `join` 메서드가 호출될 때 `Result`로 감싸져서 리턴

```rust
let numbers = Vec::from_iter(0..=1000);

let t = thread::spawn(move || {
    let len = numbers.len();
    let sum = numbers.into_iter().sum::<usize>();
    sum / len // 스레드의 리턴값
});

let average = t.join().unwrap(); // 메인 스레드에서 사용
println!("average: {average}"); // average: 500
```

⚠️ 주의

스레드에서 패닉이 일어날 수 있으므로 `unwrap`을 사용해 결과를 가져오는 것은 위험

범위 제한(scoped) 스레드

- 만일 어떤 스레드가 반드시 특정 범위에서만 존재하는 것이 확실할 경우, 지역 변수의 소유권을 "대여"할 수 있음
- 스레드의 라이프타임보다 지역 변수의 라이프타임이 더 긴 경우
- `std::thread::scope`를 사용
- 스레드에 전달되는 클로저는 지역 변수보다 먼저 사라져 `move`를 쓰지 않고도 지역 변수를 클로저에서 사용 가능

범위가 끝날 때는 실행 중인 스레드가 종료될 때까지 기다림

```rust
use std::thread;

fn main() {
    let numbers = vec![1, 2, 3];

    thread::scope(|s| {
        s.spawn(|| {
            println!("length: {}", numbers.len());
        });
        s.spawn(|| {
            for n in &numbers {
                println!("{n}");
            }
        });
    });
}
```

`numbers`에 새로운 값을 넣으려고 하면 컴파일 오류가 발생

```rust
use std::thread;

fn main() {
    let mut numbers = vec![1, 2, 3];
    thread::scope(|s| {
        s.spawn(|| {
            numbers.push(1);
        });
        s.spawn(|| {
            numbers.push(2); // Error!
        });
    });
}
```

실행 결과

```
error[E0499]: cannot borrow `numbers` as mutable more than once at a time
```

스태틱(static)

`static`은 소유권이 프로그램에 있어서 두 스레드 모두 `X`에 접근은 가능하나 소유할 수는 없음

```
use std::thread;
fn main() {
    static X: [i32; 3] = [1, 2, 3];
    {
        let t1 = thread::spawn(|| println!("{:?}", X));
        {
            let t2 = thread::spawn(|| println!("{:?}", X));
            t2.join().unwrap();
        }
        t1.join().unwrap();
    }
}
```

실행 결과

```
[1, 2, 3]
[1, 2, 3]
```

Arc(Atomic reference counting)

레퍼런스 카운팅

멀티스레딩 환경에서의 문제

1. 하나의 값을 여러 개의 스레드에서 참조하려면 반드시 값의 라이프타임이 스레드의 라이프타임 보다 길어야 함
2. 값의 소유권을 여러 스레드가 공유할 수 없기 때문에 하나의 값을 여러 개의 스레드에서 변경할 수 없음

레퍼런스 카운팅

- `Rc`를 사용해 하나의 값에 여러 개의 소유권을 생성 가능
- 하지만 `Rc`를 다른 스레드로 보내려고 하면 에러가 발생

```
use std::rc::Rc;
use std::thread;
fn main() {
    let rc = Rc::new("hello");
    let t = thread::spawn(move || {
        println!("{}", rc);
    });

    t.join().unwrap();
}
```

실행 결과

```
error[E0277]: `Rc<&str>` cannot be sent between threads safely
   --> src/main.rs:5:27
    |
5   |       let t = thread::spawn(move || {
    |                   ------------- ^------
    |                   |             |
    |  _________________|_____________within this `{closure@src/main.rs:5:27: 5:34}`
    | |                 |
    | |                 required by a bound introduced by this call
6   | |         println!("{}", rc);
7   | |     });
    | |_____^ `Rc<&str>` cannot be sent between threads safely
    |
    = help: within `{closure@src/main.rs:5:27: 5:34}`,
    the trait `Send` is not implemented for `Rc<&str>`

...
```

> `Send`와 `Sync` 트레이트는 스레드 간에 안전하게 전달할 수 있는지 여부를 결정
> 자세한 내용은 이 강의의 범위를 벗어남!

Arc(Atomic refernce counting)

- `Rc`와 동일한 기능을 제공하지만, `Arc`는 여러 스레드에서 레퍼런스 카운터를 변경하는 것이 허용
- 레퍼런스 카운터가 변경되는 작업이 아토믹하게(더 이상 쪼갤 수 없는 최소한의 연산) 수행되므로 여러 스레드에서 "레퍼런스 카운트"를 변경 가능

> ⚠️ 값을 변경할 수 있다는 것이 아님에 주의!

```rust
use std::sync::Arc;
use std::thread;
fn main() {
    let a = Arc::new([1, 2, 3]);
    let b = a.clone();
    let c = a.clone();
    let t1 = thread::spawn(move || {
        println!("Reference count: {}", Arc::strong_count(&b));
    });
    let t2 = thread::spawn(move || {
        println!("Reference count: {}", Arc::strong_count(&c));
    });
    t1.join().unwrap();
    t2.join().unwrap();
    println!("Reference count: {}", Arc::strong_count(&a));
}
```

스레드 `t1`과 `t2`의 실행 시점 차이에 따라, 실행 결과가 `3,2,1` 또는 `3,3,1`로 다를 수 있음

당연하게도, 여러 스레드에서 값을 동시에 수정하는 코드는 컴파일되지 않음!

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

fn withdraw(balance: &mut i32, amount: i32) {
    if *balance >= amount {
        thread::sleep(Duration::from_millis(10));
        *balance -= amount;
        println!("Withdrawal successful. Balance: {balance}");
    } else {
        println!("Insufficient balance.");
    }
}

fn main() {
    let mut balance = Arc::new(100);
    let t1 = thread::spawn(move || {
        withdraw(&mut balance, 50); // 🤯 가변 소유권 대여 불가!
    });
    let t2 = thread::spawn(move || {
        withdraw(&mut balance, 75);
    });
    t1.join().unwrap(); t2.join().unwrap();
}
```

동일한 파이썬 코드

```
import threading
import time

balance = 100
def withdraw(amount):
    global balance
    if balance >= amount:
        time.sleep(0.01)
        balance -= amount
        print(f"Withdrawal successful. Balance: {balance}")
    else:
        print("Insufficient balance.")

if __name__ == '__main__':
    t1 = threading.Thread(target=withdraw, args=(50,))
    t2 = threading.Thread(target=withdraw, args=(75,))
    t1.start(); t2.start()
    t1.join(); t2.join()
```

실행 결과

```
Withdrawal successful. Balance: 50
Withdrawal successful. Balance: -25
```

실행되긴 하지만 잔고가 -25가 되어버림!

뮤텍스(mutex, mutual exclusion)

- 11장에서 `Rc` 대신 `RefCell`을 사용했던 것처럼 `Arc` 대신 `Mutex`를 사용
- 뮤텍스는 Mutual exclusion(상호 배제)의 약자로, 뮤텍스는 주어진 시간에 하나의 스레드만 데이터에 액세스할 수 있도록 허용합니다. 뮤텍스를 사용하기 위해서는 두 가지 규칙을 지켜야 합니다.
- 데이터를 사용하기 전에 반드시 잠금을 해제해야 합니다.
- 뮤텍스가 보호하는 데이터를 사용한 후에는 다른 스레드가 잠금을 획득할 수 있도록 데이터의 잠금을 해제해야 합니다.

뮤텍스의 API

`withdraw`에서 `with lock`을 이용해 락이 잠금 해제 상태일 때만 해당 코드 안으로 진입

```
import time
import threading

balance = 100
lock = threading.Lock()

def withdraw(amount):
    global balance
    thread_name = threading.current_thread().name
    with lock:
        print(f"{thread_name}: Checking balance...")
        if balance >= amount:
            time.sleep(0.01)
            balance -= amount
            print(f"{thread_name}: Withdrawal successful. Balance: {balance}")
        else:
            print(f"{thread_name}: Insufficient balance.")

def check_lock():
    while lock.locked():
        print("Lock is locked.")
        time.sleep(0.01)
```

```
t1 = threading.Thread(target=withdraw, args=(50,), name="t1")
t2 = threading.Thread(target=withdraw, args=(75,), name="t2")
t3 = threading.Thread(target=check_lock)
t1.start()
t2.start()
t3.start()
t1.join()
t2.join()
t3.join()
```

실행 결과

```
t1: Checking balance...
Lock is locked.
Lock is locked.
t1: Withdrawal successful. Balance: 50
t2: Checking balance...
t2: Insufficient balance.
```

러스트

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

fn withdraw(balance: Arc<Mutex<i32>>, amount: i32) {
    let mut balance = balance.lock().unwrap(); // 뮤텍스 획득까지 기다림
    println!("{}: Checking balance.", thread::current().name().unwrap());
    if *balance >= amount {
        thread::sleep(Duration::from_millis(2));
        *balance -= amount;
        println!("Withdrawal successful. Balance: {balance}");
    } else {
        println!("Insufficient balance...");
    }
}

fn check_lock(balance: Arc<Mutex<i32>>) {
    while let Err(_) = balance.try_lock() { // try_lock()은 락이 해제될 때까지 대기하지 않음
        println!("Lock is locked.");
        thread::sleep(Duration::from_millis(1));
    }
}
```

```rust
fn main() {
    let balance = Arc::new(Mutex::new(100));

    let balance1 = Arc::clone(&balance);
    let t1 = thread::Builder::new()
        .name("t1".to_string())
        .spawn(move || {
            withdraw(balance1, 50);
        })
        .unwrap();
    let balance2 = Arc::clone(&balance);
    let t2 = thread::Builder::new()
        .name("t2".to_string())
        .spawn(move || {
            withdraw(balance2, 75);
        })
        .unwrap();
    let balance3 = Arc::clone(&balance);
    let t3 = thread::spawn(move || {
        check_lock(balance3);
    });
    t1.join().unwrap();
    t2.join().unwrap();
    t3.join().unwrap();
}
```

실행 결과

```
t1: Checking balance.
Lock is locked.
Lock is locked.
Withdrawal successful. Balance: 50
Lock is locked.
t2: Checking balance.
Insufficient balance...
```

> 참고: 락을 보유한 다른 스레드가 패닉에 빠지면 lock 호출이 실패합니다. 이때 아무도 락을 얻을 수 없게 되기 때문에 스레드에서 패닉을 발생시킵니다.

메시지 전달

MPSC(Multiple Producer Single Consumer)

- 다중 생성자-단일 소비자
- 여러 개의 스레드에서 하나의 스레드로 데이터를 보내는 방식입니다.

파이썬에서는 공식적으로 MPSC를 만드는 방법이 없기 때문에, 스레드 안정성이 보장되는 큐 자료형인 `Queue` 를 사용해 구현

```python
import threading
import time
from queue import Queue

channel = Queue(maxsize=3)

def producer():
    for msg in ["hello", "from", "the", "other", "side"]:
        print(f"Producing {msg}...")
        channel.put(msg)

def consumer():
    while not channel.empty():
        item = channel.get()
        print(f"Consuming {item}...")
        channel.task_done()
        time.sleep(0.01)

producer_thread = threading.Thread(target=producer)
consumer_thread = threading.Thread(target=consumer)
producer_thread.start()
consumer_thread.start()
producer_thread.join()
consumer_thread.join()
```

실행 결과

```
Producing hello...
Producing from...
Consuming hello...
Producing the...
Producing other...
Producing side...
Consuming from...
Consuming the...
Consuming other...
Consuming side...
```

채널은 `Sender`와 `Receiver`로 구성

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

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

    thread::spawn(move || {
        for msg in vec!["hello", "from", "the", "other", "side"] {
            let val = String::from(msg);
            println!("Producing {}...", val);
            tx.send(val).unwrap();
            thread::sleep(Duration::from_millis(10));
        }
    });

    for re in rx {
        println!("Consuming {}...", re);
    }
}
```

`try_recv` 를 사용하면 `rx` 는 송신자 `tx` 가 메시지를 전송할 때까지 기다리다가 해당 채널이 끊어지면 오류를 리턴

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

fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        let val = String::from("hello");
        thread::sleep(Duration::from_millis(1000));
        tx.send(val).unwrap();
    });

    loop {
        println!("Waiting for the signal...");
        if let Ok(received) = rx.try_recv() {  // 메세지가 올 때까지 루프 반복
            println!("Message: {}", received);
            break;
        }
        thread::sleep(Duration::from_millis(300));
    }
}
```

실행 결과

```
Waiting for the signal...
Waiting for the signal...
Waiting for the signal...
Waiting for the signal...
Waiting for the signal...
Message: hello
```

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

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

    let tx1 = tx.clone(); // 여러 개의 tx를 만들 수 있음
    thread::spawn(move || {
        for msg in vec!["hello", "from", "the", "other", "side"] {
            let val = String::from(msg);
            thread::sleep(Duration::from_millis(100));
            tx1.send(val).unwrap();
        }
    });

    thread::spawn(move || {
        let val = String::from("bye");
        thread::sleep(Duration::from_millis(1000));
        tx.send(val).unwrap();
    });

    for re in rx {
        println!("{}", re);
    }
}
```