Distributed Systems Lab 02

Concurrent computing in Rust

During the previous lab, we presented how to code basic programs in Rust and discussed some crucial features of the language, like ownership, noting that they are essential to ensuring memory safety and thread safety. This lab presents multi-threaded programming in Rust and discusses the safety features of Rust that facilitate making programs memory-safe and thread-safe. We will use files contained in this package, which you should download and extract locally.

Learning Section

Memory management

By default, values of Rust variables are stored on the stack. To safely manage data on the heap, in turn, Rust provides two special types: **`Box`** and **`Rc`**. Direct allocations on the heap are possible, but they require using *unsafe* blocks, and thus should be avoided whenever possible.

`Box<T>` is a pointer type for heap allocation. An object of type `Box<T>` owns a value of type `T` that is stored on the heap, and the object can be used as a reference (mutable or immutable) to the value. When the object goes out of scope, the value is deallocated automatically. For instance:

```
fn box_example() {
{
    let array_on_heap: Box<[u32]> = Box::new([1, 2, 3, 4, 5, 42]);
    println!("An array stored on the heap: {:?}", array_on_heap);
} // The memory is deallocated here.
```

`Rc<T>` is a reference-counting pointer. The reference-counting mechanism allows for safely obtaining multiple references to the value by cloning the reference (`Rc<T>` implements the `Clone` trait). However, it is not possible to mutate directly the value wrapped in `Rc`, as it would be unsafe to mutate a value shared via multiple references. The referenced value is deallocated when the last reference goes out of scope. For example:

```
fn rc_example() {
    let array_on_heap: Rc<[u32]> = Rc::new([1, 2, 3, 4, 5, 42]);
    let array_on_heap_2 = array_on_heap.clone();

    println!(
        "Two references to an array on the heap: {:?} and {:?}",
        array_on_heap, array_on_heap_2
    );
} // The reference counter drops to 0 here,
  // and  the memory is deallocated.
```

More advanced examples demonstrating the usage of `Box` and `Rc` are provided in file `examples/box_rc.rs` included in the package. If you know C++, `Box` is like a unique pointer whereas

`Rc` is like a shared pointer.

`Box` and `Rc` are said to be safe APIs. In the context of memory management, this means that they guarantee that no memory is freed more than once and no memory is accessed after being freed. Moreover, any memory that will not be used anymore is freed automatically. Rust achieves this by the combination of the **ownership** mechanisms, introduced during the previous lab, and the **`Drop` trait**, discussed next.

The Drop trait

During the lifetime of a process, a lot of resources need to be managed: memory, files, sockets, and so on. Some languages, C for instance, require the programmer to precisely manage the resources by hand. Other languages, like Java, implement automated garbage collection. However, such approaches are orientented mainly toward memory. There are also languages, C++ for instance, providing a way to implement destructors, which free resources when variables holding them are no longer needed. However, to this end, C++ requires a careful design of a class, for instance, disabling the copy constructor.

Rust follows the approach of C++, but the ownership mechanisms facilitate it significantly. Since each value (or each instance of some resource) has exactly one variable that owns it, the resource can be safely freed when the variable goes out of scope. This freeing is handled by the `drop()` method from the `Drop` trait, which is automatically invoked by Rust when a value is no longer needed. For instance:

```
fn drop_example() {
    {
        let int_on_heap = Box::new(42);
    } // `Box::drop()` called here and it deallocates the memory.

    {
        let vec_on_heap = Box::new(vec![1, 2, 3, 4, 5]);
        take_box_ownership(vec_on_heap);
    } // `Box::drop()` is NOT called here as the box was moved
      // out of `vec_on_heap` in the above function call.
}
```

By providing a proper implementation of the `Drop` trait, any kind of resource can be managed this way in Rust. File `examples/drop.rs` presents how this mechanism is used to close open files, and provides a simple example of a custom implementation of the `Drop` trait.

Closures

Closures are anonymous functions that can be saved in variables or passed to functions just as ordinary values. Closures can also capture values from the scope they are defined in, for example:

```
fn closure_example() {
    let inc_step = 1;
    let inc_fun = |x: u32| { x + inc_step };

    println!("inc_fun(2) = {}", inc_fun(2));
}
```

It is also possible to move values into closures, for instance:

```
fn closure_move_example() {
    let file = File::open("example.txt");
```

```
    let read_file = move || { file.read() };

    println!("File content: {}", read_file());

    // Won't compile as `file` was moved out to the closure:
    // file.rewind()
}
```

More advanced examples of closures are provided in `examples/closures.rs`.

Multiple threads of execution

Rust allows for implementing relatively easily programs with multiple threads of execution. For instance, to create a new thread, one can use the `std::thread::spawn()` function, which accepts as its argument a closure to be executed in the new thread (see `examples/threads.rs` for examples). Moreover, the safety features of Rust facilitate making the programs thread-safe. For example, consider the following problem of reference counting in two threads:

```
fn reference_counting_in_two_threads() {
    let rc = Rc::new(1);
    let rc_clone = rc.clone();
    let thread = std::thread::spawn(move || {
        let v = rc_clone.clone();
        // ...
    });
    let v = rc.clone();
    // ...

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

Without any additional synchronization, the above code might result in an invalid state of the reference counter due to the possible concurrent updates (and thus to problems like memory leaks, use after free, and so on). Fortunately, an attempt to compile the above example results in an error. This is because to ensure thread-safety, Rust introduces two special traits:

- **`Sync`**, which marks types for which it is safe to share references to them between threads.
- **`Send`**, which marks types that can be safely transferred across threads boundaries.

The compiler automatically derives the traits for custom types if they are composed only of types which are *Sync*/*Send*. It is not possible to mark a type as *Sync*/*Send* oneself without using `unsafe`. These traits are named **marker traits**, as they do not implement any methods, but they are used to declare that a type fulfills the above requirements.

The `spawn()` function accepts as its argument only closures that are *Send*, and a closure is *Send* when its environment consists only of values that are *Send*. This is not the case in the above example, because `Rc` is not *Send*, and hence, the compilation failure.

Mutex

To ensure safe accesses to some data that is shared between multiple threads, the accesses can be synchronized. Rust provides multiple synchronization primitives, for instance, the **`Mutex<T>`** type. This generic type implements mutual exclusion, and can thus be used to protect some value of type `T`. As expected, `Mutex` implements the `Sync` and `Send` traits. A simple example follows:

```rust
fn data_protected_by_mutex() {
    let data_behind_mutex = Mutex::new(vec![1, 2, 3]);
    {
        // Take the lock:
        let mut data = data_behind_mutex.lock().unwrap();

        // `Mutex::lock()` returns a wrapper around the mutable
        // reference to the value, called `MutexGuard`.

        // Safely modify the data:
        data.push(4);

    } // When the lock wrapper goes out of scope,
      // the `drop()` is called, and the lock is released.
}
```

Arc

To access a value wrapped in `Mutex` from multiple threads, a thread-safe reference is needed (`Rc` is not thread-safe). To this end Rust provides the **`Arc<T>`** type, a thread-safe pointer with an atomic counter (*Arc* stands for *Atomically Reference Counted*). Wrapping a value that is *Send* and *Sync* in the `Arc` results in a reference that is *Send* and *Sync*, as well as it can be cloned to obtain multiple references to the value (`Arc` implements the `Clone` trait). A simple example follows:

```rust
fn sharing_data_between_threads() {
    let shared_data = Arc::new(Mutex::new(vec![1, 2, 3]));
    let shared_data_copy = shared_data.clone();

    std::thread::spawn(move || {
        // Safely modify the data:
        shared_data_copy.lock().unwrap().push(4);
    });

    // Safely modify the data:
    shared_data.lock().unwrap().push(4);
}
```

Condvar

Multi-threaded programs may also require blocking threads until some event occurs. To this end, Rust provides the **`Condvar`** type which is a condition variable. A thread awaiting a shared `Condvar` is suspended, and then woken up when another thread calls the `notify_one()` or `notify_all()` method of the variable. Moreover, `Condvar` is typically paired with `Mutex`. This way a lock held by a thread is atomically released when the thread is blocked on the conditional variable, and the lock is reacquired when the thread is woken up. When using `Condvar`, spurious wake-ups are possible (i.e. wake-ups not caused by a call to `notify_one()` or `notify_all()`). A simple example follows:

```rust
fn waiting_for_event() {
    // Mutex and condition variable are typically used as a pair:
    let pair = Arc::new((Mutex::new(false), Condvar::new()));
    let pair_cloned = pair.clone();

    thread::spawn(move|| {
        let (lock, condvar) = &*pair_cloned;
```

```
8
9           // Lock the mutex:
10          let mut started = lock.lock().unwrap();
11
12          // Safely modify the data:
13          *started = true;
14
15          // Notify the other thread:
16          condvar.notify_one();
17      });
18
19      let (lock, condvar) = &*pair;
20
21      // Lock the mutex:
22      let mut started = lock.lock().unwrap();
23
24      // If the predicate does not hold...:
25      while !*started {
26          // ...wait for the notification (the thread
27          // suspened and the lock is released atomically):
28          started = condvar.wait(started).unwrap();
29
30          // Being here means holding the lock, but does not mean
31          // the predicate holds because of the spurious wake-ups!
32      }
33
34      // The lock is held, the predicate holds.
35  }
```

AtomicBool

When only simple information needs to be shared between multiple threads, Rust provides some simple thread-safe types, for instance, the **`AtomicBool`** type (there are also atomic integer types). `AtomicBool`, next to simple load and store operations, implements also more complex atomic operations like `swap()`, `compare_exchange()`, or `fetch_update()` for different atomic memory orderings. A simple example follows:

```
1   fn am_i_first() {
2       let is_first = Arc::new(AtomicBool::new(true));
3       let is_first_clone = is_first.clone();
4
5       let thread = spawn(move || {
6           is_first_clone.store(false, Ordering::Relaxed);
7       });
8
9       println!("Am I first? {}", is_first.load(Ordering::Relaxed));
10
11      thread.join().unwrap();
12  }
```

More examples with `Mutex`, `Arc`, `Condvar`, and `AtomicBool` are provided in `examples/shared_memory.rs`.

Small Assignment

Your task is to implement in Rust a thread pool. The thread pool is a structure that owns a few threads, called *workers*, and allows for submitting *tasks* that the workers execute. Every worker operates in a loop: it waits for a task, executes it, waits for the next task, and so on. The thread pool manages the workers and distributes the tasks to the workers. Multiple strategies are possible, but in the assignment you shall implement arguably the simplest one:

- There is a fixed number of threads, specified when the thread pool is constructed.
- The tasks can be executed in any order.

The thread pool shall process tasks concurrently. To synchronize the threads, we suggest using a mutex with a conditional variable.

The pool shall be recognized by Rust as *Sync*. You are not allowed to write any *unsafe* code (more precisely, you are not allowed to use the `unsafe` keyword and such a code would not compile).

Also, you shall implement the `Drop` trait for the thread pool. It shall stop all workers and wait for all threads to finish. It shall also wait until all submitted tasks are executed.

In the template for the assignment, we provide some hints, placing them in comments and in the `unimplemented!()` macros. However, they are only hints, and you are not obliged to follow them. You may implement the assignment differently as long as you do not modify the public interface.

Type of the task

In the assignment, a task is a closure of the following type:

```rust
type Task = Box<dyn FnOnce() + Send>;
```

This type specifies a closure that shall be run only once (the compiler verifies this), which is wrapped in a `Box` (so the actual closure is stored on the heap, and thus the size of a task is known at compile time), and which is marked as *Send* (the compiler verifies whether it is indeed *Send*). The `dyn` keyword means, simplifying, that objects that will be wrapped in the `Box` have to implement the `FnOnce()` and `Send` traits. However, their exact type is not specified. They can be instances of any type, as long as they implement the `FnOnce()` and `Send` traits (as opposed to, for instance, `Box<String>`, where objects wrapped in `Box` have to be instances of `String`). Since the actual type of the objects will be known only at runtime, calls to their methods have to be dynamically dispatched. (For a more detailed explanation, we recommend reading about **trait objects** in the Rust Book).

Additional Homework

Practice coding in Rust as on the next lab we will discuss and implement the first model of distributed computing. We recommend browsing the Rust Book and the Rust by Example if you have not already done it.

If you got interested in the language itself, you may want to read about other applications of the Rust programming language. For example, about programming embedded systems, like microcontrollers, in The Embedded Rust Book, or about developing fast and reliable code for the Web in Rust and WebAssembly.

Authors: F. Plata, K. Iwanicki, M. Banaszek, W. Ciszewski, M. Matraszek.