COMP 737011 - Memory Safety and Programming Language Design

Lecture 8: Rust Concurrency

徐 辉
xuh@fudan.edu.cn

Module std::sync

Modules

atomic	Atomic types
mpsc	Multi-producer, single-consumer FIFO queue communication primitives.

Structs

Arc	A thread-safe reference-counting pointer. 'Arc' stands for 'Atomically Reference Counted'.
Barrier	A barrier enables multiple threads to synchronize the beginning of some computation.
BarrierWaitResult	A `BarrierWaitResult` is returned by `Barrier::wait()` when all threads in the `Barrier` have rendezvoused.
Condvar	A Condition Variable
Mutex	A mutual exclusion primitive useful for protecting shared data
MutexGuard	An RAII implementation of a "scoped lock" of a mutex. When this structure is dropped (falls out of scope), the lock will be unlocked.
Once	A synchronization primitive which can be used to run a one-time global initialization. Useful for one-time initialization for FFI or related functionality. This type can only be constructed with `Once::new()`.
OnceState	State yielded to `Once::call_once_force()`'s closure parameter. The state can be used to query the poison status of the `Once`.
PoisonError	A type of error which can be returned whenever a lock is acquired.
RwLock	A reader-writer lock
RwLockReadGuard	RAII structure used to release the shared read access of a lock when dropped.
RwLockWriteGuard	RAII structure used to release the exclusive write access of a lock when dropped.
WaitTimeoutResult	A type indicating whether a timed wait on a condition variable returned due to a time out or not.
Weak	`Weak` is a version of `Arc` that holds a non-owning reference to the managed allocation. The allocation is accessed by calling `upgrade` on the `Weak` pointer, which returns an `Option<Arc<T>>`.

https://doc.rust-lang.org/std/sync/index.html#structs

Module std::marker

Structs

PhantomData	Zero-sized type used to mark things that "act like" they own a `T`.
PhantomPinned	A marker type which does not implement `Unpin`.

Traits

DiscriminantKind Experimental	Compiler-internal trait used to indicate the type of enum discriminants.
StructuralEq Experimental	Required trait for constants used in pattern matches.
StructuralPartialEq Experimental	Required trait for constants used in pattern matches.
Unsize Experimental	Types that can be "unsized" to a dynamically-sized type.
Copy	Types whose values can be duplicated simply by copying bits.
Send	Types that can be transferred across thread boundaries.
Sized	Types with a constant size known at compile time.
Sync	Types for which it is safe to share references between threads.
Unpin	Types that can be safely moved after being pinned.

https://doc.rust-lang.org/std/marker/index.html

Outline

- 1. Basic features
  a) Atomic types
  b) ARC/Weak
  c) Memory barrier
  d) Locks: Mutex/RwLock
  e) Conditional variable
  f) Once
  g) mpsc
- 2. Marker Trait
  - Send and Sync

1. Basic features

a) Atomic Types

- Several atomic types
  - AtomicBool,
  - AtomicIsize,
  - AtomicUsize,
  - …
- Similar to C++ std::<atomic>

```
let mut foo = AtomicI32::new(0);
*foo.get_mut() = 5
foo.fetch_add(10, Ordering::SeqCst);
foo.compare_and_swap(5, 10, Ordering::Relaxed)
```

assignment

atomic add

CAS

b) Arc<T>: Atomically Ref Counted

- Similar to RC<T>, but is thread safe
- Use atomic operations for reference counting
- Mutating through an Arc generally use Mutex, RwLock, etc.

```
fn main() {
    let v = Arc::new(Mutex::new(vec![1,2,3]));
    for i in 0..3 {
      let cloned_v = v.clone();
      thread::spawn(move || {
         cloned_v.lock().unwrap().push(i);
      });
    }
}
```

move the ownership to
the thread closure

c) Memory Barrier

- Ordering is an enumerate type
  - SeqCst: sequential consistency
  - Acquire-Release: commonly used when implementing locks
    - Acquire: no read/write after the load is reordered before
    - Release: no read/write before is reordered after this store
  - Relaxed: no restriction

Example of a simple mutex lock

```
pub struct Mutex { flag: AtomicBool, }
impl Mutex {
    pub fn new() -> Mutex {
        Mutex { flag: AtomicBool::new(false), }
    }
    pub fn lock(&self) {
        while self.flag.compare_exchange_weak(false, true,
            Ordering::Relaxed, Ordering::Relaxed).is_err()
              {}
        fence(Ordering::Acquire);
    }
    pub fn unlock(&self) {
        self.flag.store(false, Ordering::Release);
    }
}
```

d) Mutex

- Use lock() or try_lock() to access the data
  - Returns Result<T>
  - lock() is blocking mode
    - most usage simply unwrap() the result, why?
  - try_lock() is nonblocking mode
    - returns Err() if fails

```
let arc = Arc::new(Mutex::new(0));
let arc1 = arc.clone();

let _ = thread::spawn(move || -> () {
    let mut data = arc.lock().unwrap();
    *data += 1;
}).join()
```

d) Mutex: Poison Strategy

- The lock could be poisoned if a thread holding the lock panics
- Use into_inner()

```
let arc = Arc::new(Mutex::new(0));
let arc1 = arc.clone();

let _ = thread::spawn(move || -> () {
    let mut data = arc1.lock().unwrap();
    panic!();
}).join()

assert_eq!(arc.is_poisoned(), true);
let mut guard = match arc.lock() {
    Ok(guard) => guard,
    Err(poisoned) => poisoned.into_inner(),
};
*guard += 1;
```

The lock is poisoned

e) Condition Variable

- Synchronizing primitive used when threads need to wait for a resource to become available.

```
let pair = Arc::new((Mutex::new(false), Condvar::new()));
let pair2 = Arc::clone(&pair);
thread::spawn(move|| {
    let (lock, cvar) = &*pair2;
    let mut started = lock.lock().unwrap();
    *started = true;
    cvar.notify_one();
});
let (lock, cvar) = &*pair;
let mut started = lock.lock().unwrap();
while !*started {
    started = cvar.wait(started).unwrap();
}
```

f) Once

- A synchronized primitive to run global initialization only one time
  - access `static mut` variables.

```
static mut VAL: usize = 0;
static INIT: Once = Once::new();

fn get_cached_val() -> usize {
    unsafe { INIT.call_once(|| {
        VAL = expensive_computation();
        });
        VAL
    }
}
```

g) mpsc

- Multi-producer, single-consumer FIFO queue communication primitives
  - Asynchronous mode,
  - synchronous mode

```
let (tx, rx) = channel();
//let (tx, rx) = sync_channel()
for i in 0..10 {
    let tx = tx.clone();
    thread::spawn(move|| { tx.send(i).unwrap(); });
}
//drop(tx); stop rx waiting
while let Ok(msg) = rx.recv()
    let j = rx.recv().unwrap();
    assert!(0 <= j && j < 10);
}
```

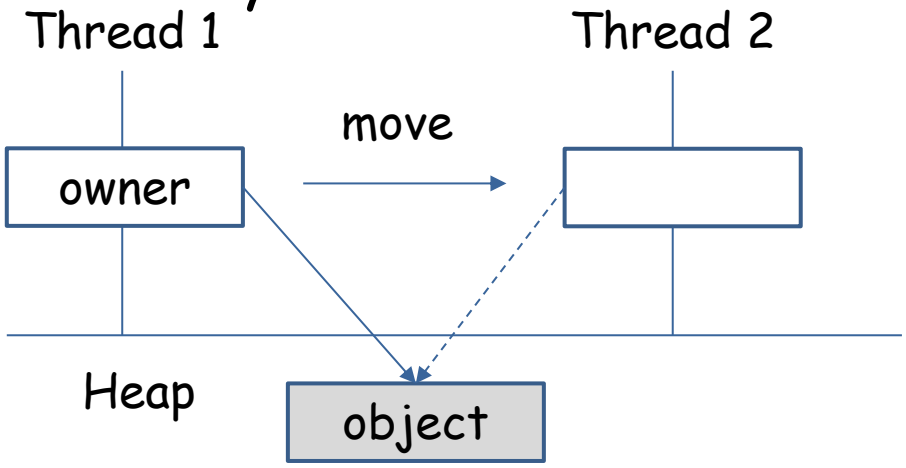
2. Marker Trait

Marker Traits

- Marker Traits have no methods to implement.
  - copy
  - sized
  - send
  - sync
  - unpin

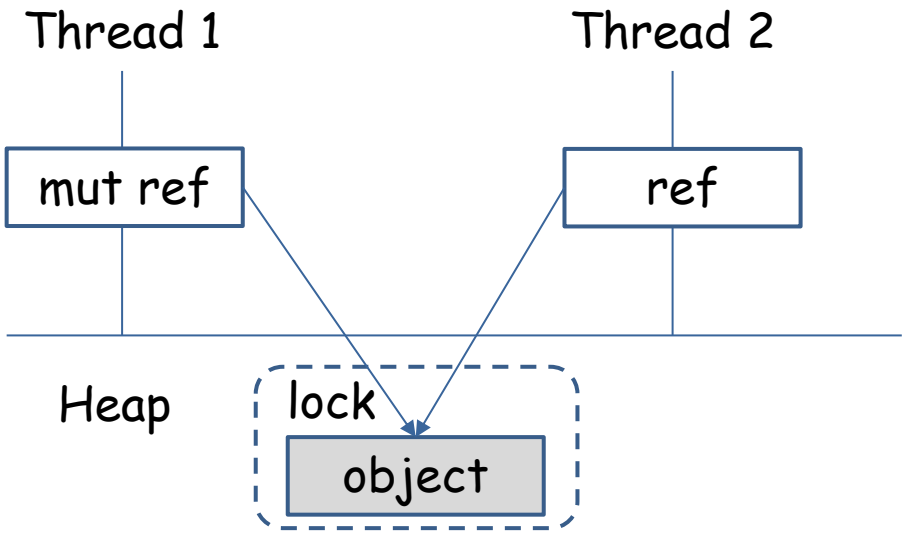
Send

- The ownership of values of the type implementing Send can be transferred between threads.
- Use the move operator, which is similar as =
  - For types of Copy trait, make a copy of the object
  - For non-copy, transfer the ownership
- Almost all primitive types are Send
- Any struct composed of Send types is automatically marked as Send



Sync

- The type implementing Sync is safe to be referenced from multiple threads
- Any type T is Sync if &T is Send
- Sync is usually more rigid than Send
  - Are there any examples that are Sync but not Send?
    - cases are rare
    - exceptions may relate to thread-local features,
      - e.g., MutexGuard



Types Are Not Send

- Raw pointers are neither Send nor Sync
  - Possible to create shared objects (although unsafe)
  - Developers should to tell compiler it is safety
- RC<T> cannot be Send, why?
  - Shared ownership
  - Problems in concurrent reference counter update
- RefCell/Cell are Send but not Sync
  - unsynchronized interior mutability

```
impl<T> !Send for Rc<T>
impl<T> !Sync for Rc<T>
```

Can Cell be Send/Sync

```
fn check_sync<T: Sync>(_: T) {}
fn check_send<T: Send>(_: T) {}

fn testCell(){
    let mut v = Cell::new(1);
    check_send(v);
    //check_send(&mut v);
    //check_send(&v);
    //check_sync(&v);
    //check_sync(&mut v);
}
```

success

success

fail

fail

fail

Can Mutex be Send/Sync

- only if T is Send

```
fn testMutex1(){
    let mut v = Mutex::new(1);
    //check_send(v);          success
    //check_send(&v);         success
    //check_send(&mut v);     success
    //check_sync(&v);         success
    check_sync(&mut v);       success
}
```

```
fn testMutex2(){
    let mut v = Mutex::new(Cell::new(1));
    //check_send(v);          success
    //check_send(&v);         success
    //check_send(&mut v);     success
    //check_sync(&v);         success
    check_sync(&mut v);       success
}
```

Can Mutex be Send/Sync

```
fn testMutex3(){
    let mut v = Mutex::new(&Cell::new(1));
    //check_send(v);
    //check_sync(v);
}
```

fail

fail

```
fn testMutex4(){
    let mut cell = Cell::new(1);
    let mut v = Mutex::new(&mut cell);
    check_send(v);
    //check_sync(&v);
}
```

success

success

Questions

- Does ARC<T> have bound on T to be thread-safe?
  - T does not need to be Send + Sync when constructing ARC<T>
  - The compiler checks the wrapped data during compilation

```
fn testArc(){
    let mut cell = Cell::new(1);
    let mut v = Arc::new(cell);
    let v1 = v.clone();
    thread::spawn(move || {
        (*v1).set(3);
    }).join();
    (*v).set(2);
}
```

Compilation error

Implementing Send/Sync is Unsafe

```
struct Unsend{ ptr: *mut i64, }
impl Unsend{
    fn add(&self, i:i64){
        unsafe{*(self.ptr) = *self.ptr + i};
    }
}
unsafe impl Send for Unsend{}
unsafe impl Sync for Unsend{}

fn main(){
    let mut var = 0i64;
    let mut v = Unsend{ptr:&mut var as *mut i64};
    let tid = thread::spawn(move || {
        for i in 1..100001{
            v.add(i);
        }
    });
    for i in 1..100001{
        var+=i;
    }
    tid.join();
    println!("{}",var);
}
```

In-Class Practice

- Rewrite your program (binary search tree or double-linked list) to be thread-safe
  - Support Sync/Send

More Reference

- https://doc.rust-lang.org/nomicon/send-and-sync.html
- https://nyanpasu64.github.io/blog/an-unsafe-tour-of-rust-s-send-and-sync/
- https://cseweb.ucsd.edu/classes/sp17/cse120-a/applications/ln/lecture7.html