COMP 737011 - Memory Safety and Programming Language Design

Lecture 9: Rust Concurrency

徐 辉
xuh@fudan.edu.cn

Outline

1. Multi-Thread Rust
2. Basic Concurrency APIs
3. Marker Trait for Concurrency

1. Multi-Thread Rust

Spawn A New Thread

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

fn main() {
    let thd = thread::spawn(|| {
        ...
        thread::sleep(Duration::from_millis(1));
    });
    thd.join().unwrap();
}
```

spawn a thread

wait for the thread ends

Spawn A Group of Threads

```
let mut threads = vec![];

for i in 0..3 {
    threads.push(thread::spawn(move || {
        ...
    }));
}

for t in threads {
    t.join();
}
```

save thread handlers in a vector

Access Objects In the New Thread

- Access the same object from multiple threads is risky:
  - race condition
  - the thread may outlive the lifetime of the object

```
let mut x = 1;
//let tid = thread::spawn(|| {
let tid = thread::spawn(move|| {
    x = 10;
    println!("spawn: x = {}", x);
});
tid.join().unwrap();
println!("main: x= {}", x);
```

move the ownership or copy

copied x

```
spawn: x = 10
main: x= 1
```

Object Access: Cont'd

```
let mut x = Box::new(1);
let tid = thread::spawn(move|| {
    *x = 10;
    println!("spawn: x = {}", x);
});
tid.join().unwrap();
println!("main: x= {}", x);
```

move the ownership of x to the thread

Illegal to access x

```
let mut x = Box::new(1);
let mut y = x.clone();
let tid = thread::spawn(move|| {
    *y = 10;
    println!("spawn: y = {}", y);
});
tid.join().unwrap();
println!("main: x= {}", x);
```

make a clone of x as y

access y

```
spawn: y = 10
main: x= 1
```

Share Data Among Threads

```
let x = Box::new(1);
for i in 0..3 {
    let r = &x;
    thread::spawn(move || {
        println!("{:?}",r);
    });
}
```

the thread may live longer than x

```
let x = RC::new(Box::new(1));
for i in 0..3 {
    let cl = x.clone();
    thread::spawn(move || {
        println!("{:?}",cl);
    });
}
```

RC is not thread safe

We Need Concurrency-Safety APIs

- Basic APIs
  - Atomicity or lock
  - Synchronization or memory Barrier
- Advanced features

2. Basic Concurrency APIs

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

Sample Mutex Lock with Memory Barrier

```
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);
    }
}
```

all subsequent loads will see the stored data

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.

```
let x = Arc::new(Box::new(1));
for i in 0..3 {
    let cl = x.clone();
    thread::spawn(move || {
        println!("{:?}",cl);
    });
}
```

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 x = Arc::new(Mutex::new(0));

for i in 0..3 {
    let cl = x.clone();
    thread::spawn(move || {
        let mut data = cl.lock().unwrap();
        *data += 1;
        println!("{:?}", data);
    });
}
```

Do not need to unlock, why?

Synchronizing Primitive: Condition Variable

- Do not consume CPU when threads need to wait for a resource to become available
- How to implement the feature?

```
let x = Arc::new((Mutex::new(0), Condvar::new()));
let cl = Arc::clone(&x);
thread::spawn(move|| {
    let (l, c) = &*cl;
    let mut t = l.lock().unwrap();
    *t = 100;
    c.notify_one();
});
let (l, c) = &*x;
let mut t = l.lock().unwrap();
while *t == 0 {
    t = c.wait(t).unwrap();
    println!{"while: t = {}", t};
}
```

Mutex: Poison Strategy

- What if a thread holding the lock panics?
- Using a poison flag to detect/recover from the bad state

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

let _ = thread::spawn(move || -> () {
    let mut data = cl.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;
```

Panic the thread

The lock is poisoned

Release the locked data

Message Passing

- Multi-producer, single-consumer FIFO queue
  - Asynchronous or synchronous mode

```
use std::sync::mpsc;
use std::thread;
fn main() {
    let (tx, rx) = mpsc::channel();
    let tx = tx.clone();
    let tid = thread::spawn(move|| {
        for i in 0..10 {
            tx.send(i).unwrap();
        }
    });

    while let Ok(msg) = rx.recv(){
        println!{"receive: {}",msg};
    }
}
```

Synchronizing Primitive: Once

- 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
    }
}
```

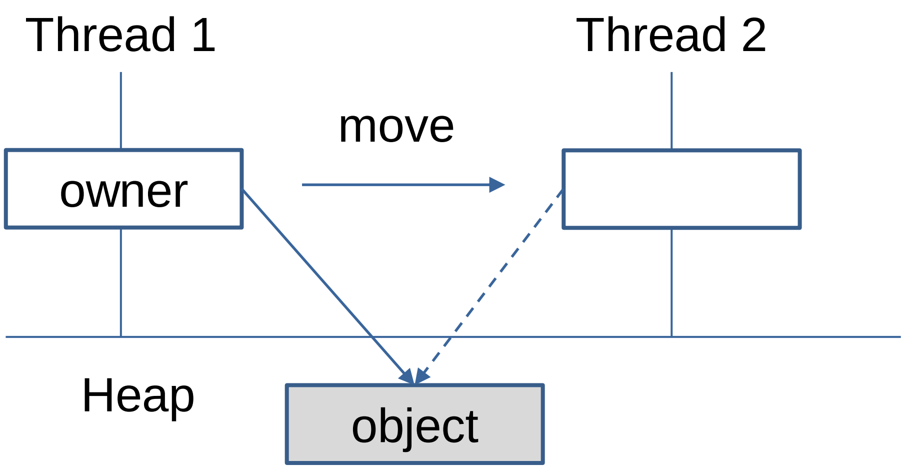
3. Marker Trait for Concurrency

Marker Traits for Concurrency

- Marker Traits have no methods to implement
- They are compiler intrinsic and auto derived
  - Send/!Send
  - Sync/!Sync
- Other marker traits
  - Copy/!Copy
  - Sized/!Sized
  - Unpin/!Unpin

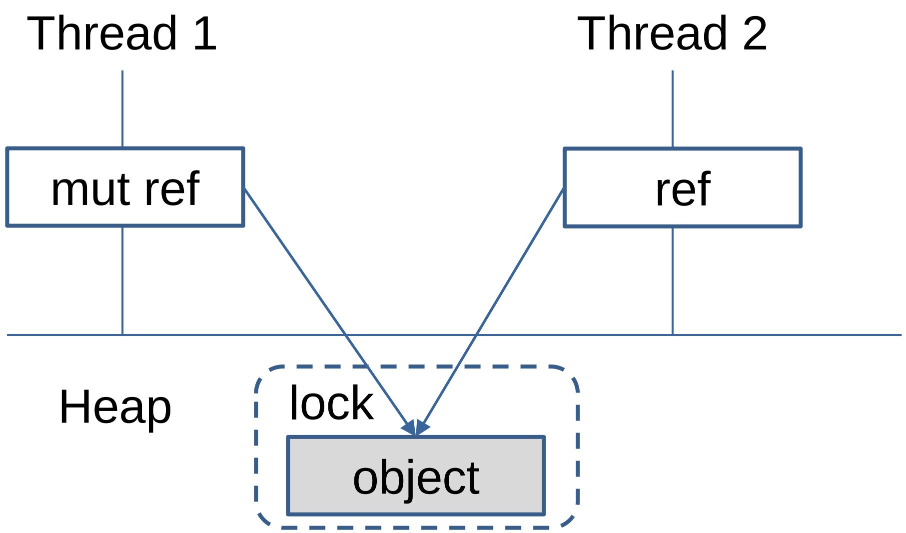
Send

- The type of 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 of 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. Why?



Raw pointers are neither Send nor Sync

- Possible to create shared objects (although unsafe)
- Should be manually implemented as unsafe

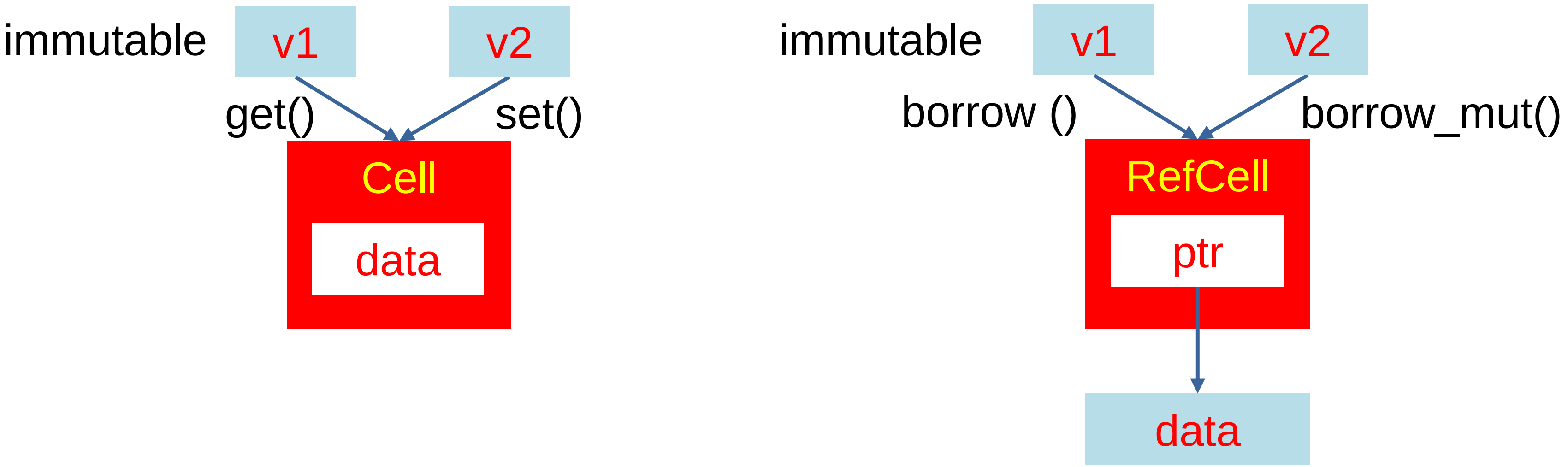
```
struct Unsend{ ptr: *mut i64, }
impl Unsend{
    fn add(&self, i:i64){
        unsafe{*(self.ptr) = *self.ptr + i};
    }
}
unsafe impl Send for Unsend{}
unsafe impl Sync for Unsend{}

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);
```

Implement Send/Sync is unsafe

Can Cell/RefCell Be Send/Sync?

- unsynchronized interior mutability
- Send but not Sync



Code to Verify Send/Sync Properties

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

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

success
success
fail
fail

Rc<T> and Arc<T>

- Rc<T> is neither Send nor Sync, why?
  - !Sync: atomicity in reference counter update
  - !Send: cloned Rc exist in multiple threads
- Does Arc<T> have bound on T to be thread-safe?
  - The compiler checks the wrapped data during compilation

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

Force Rc<T> to be!Send + !Sync

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

No Bound on T

Compilation error

Can Mutex be Send/Sync?

- Require T is Send

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

```
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
```

Can Mutex be Send/Sync? Cont'd

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

fail
fail

```
Let mut cell = Cell::new(1)
let mut v = Mutex::new(&mut cell);
check_send(v);
//check_sync(v);
```

success
success

Sync but not Send?

- Cases are rare
- Exceptions may relate to thread-local features,
  - e.g., MutexGuard

In-Class Practice

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