1 Basic Concurrency Primitives, Ch1

topics: interior mutability, threadsafety, runtime borrow check

1.1 single thread interior mutability

RefCell: borrow at runtime
Cell: value replacement, not borrow; limited to word size

1.2 threadsafe interior mutability

Mutex: exclusive borrow at runtime
RwLock: differentiates borrow type: exclusive vs. shared read only
Atomics: value replacement, not borrow; limited to word size

UnsafeCell: express raw pointer to wrapped data via unsafe block; in practice wrapped by a safer interface to user

Traits for threadsafety: Send, Sync
T: Send $\iff$ T can be transferred to another thread
T: Sync $\iff$ T can be shared with > 1 threads; &T: Send
all primitve types are Send + Sync

auto traits:

- automatically opt-in
- manually opt-out
- recursively deduced on filds of structs

un-implemented types:

```
Cell<T>: Send + Sync!
* const T / * mut T: !Send + !Sync
Rc<T>: !Send + !Sync
std::marker::PhantomdData<T> where T: !Send / !Sync
```

force opt-in for un-implemented type:

```
unsafe impl Send/Sync for T {}
```

1.3 Mutex

T is usually Send(not required) in which case the Mutex gives Sync:
T: Send $\implies$ `Mutex<T>: Sync`

logical states: unlocked, locked

owning wrapper over T

interface makes access to T safer

MutexGuard as proof of exclusive access; drop automatically triggers unlock at the end of its lifetime

efficient usage: make locked interval as short as possible

lock poisoning: when thread panics while holding the lock

- lock is released
- the invoking method call errors
- further invoking a poisoned mutex returns error and also a locked MutexGuard in case user can correct it to some consistent state

unnamed guard may not be immediately dropped in certain statements:

```
if ... /*dropped here; only boolean value needed*/ {
  ...
}

if let ... = ... {
  ... /*dropped here; may borrow from let expression*/
}
```

1.4 ReaderWriterLock

requires T: Send + Sync:
T: Send + Sync $\implies$ `RwLock<T>: Send + Sync`

logical staes: unlcoked, locked by 1 exclusive accessor, locked by any number of shared readers

differentiating lock guards:
read() $\implies$ ReLockReadGuard: Deref
write() $\implies$ ReLockWriteGuard: DerefMut

writer starvation issue to cosider for fairness of access

1.5 Thread Signaling

1.6 park/unpark

park: current thread put itself to sleep
unpark: another thread wakes sleeping thread; needs handle of the sleep read from spawn() method or thread::current()

spurious wakeup due to false sharing, etc. $\implies$ user provide a check upon wakeup

request to unpark recorded if unpark happens before park in order to avoid lost notification, but does not stack up (max of 1 unpark recorded)

1.7 Condition Variable

signaling events related ro protected data of mutex

methods: wait, notify

atomically unlock mutex and start waiting (to avoid lost notification)

1.7.1 Communication

waiting thread:
takes MutexGuard as input unlocks mutex thread put to sleep thread wakes (via a notify of CondVar or spurious wakeup) relocks mutex and returns MutexGuard

notifying thread:
invoke notify on CondVar

1.7.2 Spurious Wakeup

need additional memory to check actual event: can add this along with the original value wrapped by mutex
use a loop with wait to put thread back to sleep if condition not met

usage: 1 CondVar for 1 mutex

optionally can wait with timeout parameter to unconditionally wakeup thread after timeout

1.8 Comparison of Interior Mutability Primitives

	value replacement	reference / borrow
1 thread	Cell	RefCell
threadsafe	Atomic	Mutex/RwLock

1.9 Comparison of Shared Ownership Primitives

Rc/Arc: act similar to Box / smart pointer but with dropping logic to take care of deallocation for shared data

1 thread	Rc
threadsafe	Arc

1.10 Traits for Interior Mutability Primitives

`T: Send` $\implies$ `Cell<T>: Send + !Sync` (usual practical case)
`T: !Send` $\implies$ `Cell<T>: !Send + !Sync`
`T: Send` $\implies$ `RefCell<T>: Send + !Sync` (usual)
`T: !Send` $\implies$ `RefCell<T>: !Send + !Sync`
`T: Send` $\implies$ `Mutex<T>: Send + Sync` (usual)
`T: !Send` $\implies$ `Mutex<T>: !Send + !Sync`
`T: Send + Sync` $\implies$ `RwLock<T>: Send + Sync` (usual)
`T: !Send / !Sync` $\implies$ `RwLock<T>: !Send + !Sync`

1.11 Traits for Shared Ownership Primitives

`Rc<T>: !Send + !Sync`
`T: Send + Sync` $\implies$ `Arc<T>: Send + Sync`

1.12 Typical Usage Pattern

`Arc<Mutex<T>>`
where:
Arc allows threadsafe immutable sharing
Mutex allows interior mutability using references across ≥ 1 threads

`Rc<RefCell<T>>`
where:
Rc allows single thread immutable sharing
RefCell allows interior mutability using references in single thread

`Rc<Cell<T>>`
where:
Rc allows single thread immutable sharing
Cell allows interior mutability using value in single thread

`Arc<Atomic<T>>`
where:
Arc allows threadsafe immutable sharing
Atomic allows interior mutability using value across ≥ 1 threads

2 Atomics

`fetch_and_modify`
`swap`
`compare_exchange`:

- ABA problem for pointer algorithms
- weak version exists for more efficient impl. on some hardware at expense of spurious wakeup

`fetch_update` $\iff$ load followed by loop with `compare_exchange_weak` and user provided computation

2.1 Scoped Thread

regular `std::thread::spawn` requires closure to be Send $\implies$ all captures of closure are required to be Send

`std::thread::scope`:
borrows object of non-static lifetime that can outlive thread
muatiblity rules apply
threads are automatically joined at the end of the scope

2.2 Lazy Initialization

execute once by 1 thread, sharable afterwards
race possible from threads, but this is different from data race which causes undefined behaviour
can use `CondVar` / thread parking / `std::sync::Once` / `std::sync::OnceLock` to avoid wasted compute from multiple threads

2.3 Move Closure

transfer ownership of value
capture variable via copying/moving instead of borrowing
copying reference in a move closure in order to borrow from variable
note: Atomic does not implement Copy trait

2.4 Data Sharing Between Threads in General

data shared need to outlive all involved threads:

- make data owned by entire program via static lifetime (static item exists even before start of the main program
- leak an allocation and promise never to drop it from that point onward in the duration of the entire program: eg: `Box::leak(Box::new(..))`
  note: `'static` means the object will exist until the end of the program but may not exist at the start of the program
  note: Copy *implies* when moved, the original value still exists
- reference counting: track ownership and drop when no owners
  eg: `std::rc::Rc`: clone increments counter only and gives reference to allocation
  eg: `std::sync::Arc`: version of *Rc* that is safe between threads

use of scope and variable shadowing to reuse identifiers when cloning:
shadowing: original name is not obtainable anymore in current scope
original name still obtainable in an outerscope, can clone it in another inner scope

reference counted pointers(`Rc` and `Arc`) have same restrictions as immutable reference (`&T`)

mutable borrows are guaranteed at compile time $\implies$ mutable aliasing between 2 variables does not occur; optimization to remove impossible code blocks possible

assumptions held by the compiler:

- an immutable reference exists *implies* no other mutable references to the associated data exists
- there is at maximum 1 mutable reference to an object at anytime

if such assumptions are broken, then undefined behaviour exists: more wrong conclusions may be propagated through optimizations

`unsafe` blocks are also assumed to be sound by the compiler which means compiler may apply optimizations and elide code when feasible

2.5 Interior Mutability

shared reference `&T`: copied and sharable (not mutable)
exclusive reference `& mut T`: exclusive borrow of T

interior mutability provides more flexibility for shared data that needs mutation

`Cell / Atomic`: replace value, no borrow
`RefCell / Mutex`: runtime borrowing; book-keeping cost for existing borrows; failable at runtime