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 (UB)
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 UB 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

3 Memory Ordering

defining happens-before relations across threads

concurrent non-atomic stores to same variable causes data race $\implies$ UB

lack of globally consistent order

thread spawn/join: automatically enforces happens-before relation

note: current theoretical model for formalizing memory ordering bug: cyclic reasoning / value out of thin air

3.0.1 Relaxed Ordering

- per atomic variable: a total modification order in every run of the program $\implies$ all modifications of the said atomic variable happen in 1 order that is consistent/same from views of every thread
- multiple possible orderings may exist when the program is run multiple times, but each run satisfies a total modification order
- no happens-before relation

3.0.2 Release-Acquire Ordering Pair

pairing:
store operation specified with release semantics
load operation specified with acquire semantics

happens-before relation formed at runtime when load succeeds: all memory operations before release store is observable by and after acquire load

release store of an atomic variable may be modified by any number of fetch-modify / compare-exchange operations and still have a happens-before relation with an acquire load afterwards on the said atomic variable

any store of the associated atomic variable breaks the chain of a release-acquire pair (that previously starts with a release store and possibly followed with fetch-modifies/compare-exchanges)

use of non-atomic variable in different threads and borrow checker $\implies$ may need unsafe blocks

3.0.3 Release-Consume Ordering Pair

pairing:
store operation specified with release semantics
load operation specified with consume semantics

happens-before relation for associated atomic variable in the release store and the dependent expressions in the consumer thread

practically, hard to define dependent evaluation and implementation tends to fallback to acquire semantics instead

3.0.4 Sequentially Consistent Ordering

pairing:
store operation specified with SeqCst semantics
load operation specified with SeqCst semantics

guarantees of:

- acquire ordering
- release ordering
- globally consistent ordering of all SeqCst operations (every SeqCst operation in a program is a part of a single total order that all threads agree on)

can replace acquire and release ordering and maintain happens-before relation

3.1 Memory Fence

separate memory ordering semantics from atomic operations

it can take place of acquire / release / other memory order operations

types of fences:

- release fence
- acquire fence
- acquire-release fence
- sequentially consistent fence

3.1.1 Practical Replacement

without fences	**with fences**
release store	fence with release ordering ... atomic store (any memory ordering)
acquire lead	atomic load (any memory ordering) ... fence with acquire ordering

any atomic store following release fence is observable by any atomic load before acquire fence $\implies$ happens-before relation is established between the release-acquire fences pairing

3.1.2 Practical Usages

- can be used for multiple variables at once
- conditional fence (apply happens-before relation only after certain condition is met)
  eg: place acquire fence in conditional branch that succeeds that is relevant to the atomic variable

```
let p = var.load(relaxed);
if p == ... {
  fence(acquire);
  do_something(...);
}
```

- may be more efficient if atomic variable is expected to fail in comparison often (let atomic variable be loaded with relaxed memory ordering)

3.1.3 SeqCst Fence

- is both a release fence and an acquire fence
- is part of a single total order of sequentially consistent operations

3.2 Compiler Fence

does not prevent processor from reordering instructions

Rust compiler fence: `std::sync::atomic::compiler_fence`

uses:

- process-wide memory barriers
- special cases of signal handler/interrupt

3.3 FAQs

memory model is not related to timing

memory model defines order of operations and affects instruction reordering

SeqCst implies the operation depends on the total order of every single SeqCst operation in the program
$\Longrightarrow$ usually overly tall claim
$\Longrightarrow$ more relaxed constraints may be easier to review (eg: release-acquire pairs)

release store not form happens-before relation with SeqCst store: for a part of a globally consistent order, both operations need to be SeqCst

3.4 Summary

each atomic variable has its own total modification order that all threads agree on

single thread: happens-before relations exist between every single operations

unlocking a mutex happens-before locking that mutex

SeqCst results in 1 globally consistent order of operations that participates in SeqCst, but it is usually overly constraining

fences allow combining memory ordering of multiple operations for efficiency or applying conditional memory ordering for efficiency

happens-before relation exist when:

- threads spawn / join
- acquire load from a release store on an atomic variable
- fetch-modifies / compare-exchanges in between a release-acquire pair on an atomic variable is still valid for that happens-before relation

4 Processor

5 OS Primitives

6 Primitive Implementation Examples

6.1 Spin Lock

release store (unlock) and acquire load (lock) pair for prevention of data race (UB)

`std::hint::spin_leep()`: possible optimization of processor

possible implementation:

- wraps actual data inside an `UnsafeCell<T>` for interior mutability
- requires `T: Send`
- locking provides exclusive access (and provides `Sync` trait)
- uses `unsafe` blocks in function, user will not have to use `unsafe`
- use lock guard (representing safe access to locked data) pattern to managae lifetime of locked access to protected data:
  - implement `Deref`, `DerefMut` to access data for user ergonomics (behave similar to reference)
  - implement `Drop`: automatic release store (unlocking) when lifetime of the guard ends
  - manual drop also possible: this consumes and ends the valid lifetime of the guard and hence access to data at compile time $\implies$ any further reference and borrow to guard is invalid and will be flagged as error by the compiler

6.2 Channels

6.2.1 One Shot Channel

- 1 message only from 1 thread to another thread
- `T: Send`
- use of `unsafe`:
  - may be unitialized
  - non-copy data must not be duplicated
  - manual content drop may be necessary: leaking/forgetting is safe but sometimes undesirable
  - eg: `std::mem::MaybeUninit<T>` (unsafe version of `Option<T>`) for efficiency where user tracks its initialized status
  - use `UnsafeCell`'s interior mutability for sharing
  - wrapping shared struct `Channel` requires `T: Send` and gives `Sync` in return
- use of atomic swaps for setting one time flags
- encoding of multiple states in one word and atomic compare-exchanges
- possible use of runtime check/panic instead of letting UB happen
- use type checking from compiler to avoid errors: move/consume to avoid unwanted reuse of resources:

  use of non-Copy type and pass by value / consume by called function $\implies$ prevent caller from using that object again at compile time (elides some runtime checks)

  - `TX-RX` pair for message passing, `Channel` shared inside their private implmentations
  - use of `Arc<T>` for sharing of allocation and resource dropping: `RX` drop and `TX` drop $\implies$ `Arc<T>` drop $\implies$ `T` drop
  - `Arc<T>` incurs extra runtime overhead for allocation
- allocation optimization
  - borrowing instead of memory allocation (`Arc`)
  - use of lifetimes and mutable borrow for compile time checks
  - `Channel` explicitly created by user ahead of time and passed in to `RX` and `TX` as references upon construction in the `split` method
  - `TX, RX` take in additional lifetime parameter which is the same as the lifetime of the borrowed `Channel`: when `TX` or `RX` is present, existing `Channel` cannot be mutably borrowed again until `TX` and `RX` are both dropped
  - `Channel` needs to outlive `TX` and `RX` for compiler check to pass
  - `Channel` resets its contents on entry to `split` method in case it is used multiple times
- overwriting content for fresh initialization: after 1st borrow expires, subseqquent borrows are made on these newly created resources
- blocking interface:
  - make `RX` object not `Send`, such as using a `PhantomData<* const ()>` member field so that auto trait deduction for the wrapping struct is propagated to be `!Send`:
    `* const () : !Send` $\implies$
    `PhantomData<* const ()> : !Send` $\implies$
    wrapping struct is `!Send`
  - `RX` stays on the same thread, `TX` allows to cross thread boundaries
  - use receiving thread's handle to invoke waking up a blocked thread (place this inside the sender struct)
  - call unpark on receiving thread's handle after release store optation is performed by the sender
  - recever checks `Channel`'s variable to avoid spurious wakeup

6.3 Arc

basic implementation: use a pointer type to the shared underlying data via `std::ptr::NonNull`, use counter info for the shared data; use `NonNull::from(Box::leak(Box::new(..)))` to get a pointer from initial allocation

implement ergonomic methods `deref` and `deref_mut`

let `cloning` change internal counter and point to the shared data

require `T: Send + Sync` and give wrapping `Arc<T>` `Send + Sync`

auto traits is not active for raw pointer types (including `NonNull`) wrt. `Send` and `Sync`

when only find decrement needs to be acquire and all others release:

```
if var.fetchsub(1, release) == 1 {
  fence(acquire);
}
```

exclusive access to shared data, conditionally in a runtime branch, eg:

```
fn get-Mut(arc: & mut Self) -> Option<& mut T>
```

miri interpreter for simulation and verification of unsafe code

weak version of `Arc<T>`: `Weak<T>`

- `T` can be shared between `Arc<T>` and `Weak<T>`
- `Weak<T>` does not prevent drop of `T`, eg: all `Arc<T>` dropped $\implies$ `T` dropped
- `Weak<T>` exists without reliance of `T`, which can provide conditional access to `& T`-like object:
  implement upgrade function to get `Option<Arc<T>>` where it's `None` if `T` is already dropped

cycle breaking: use 2 counters:

- strong pointer count (`data_ref_count`)
- weak pointer + strong pointer count (`alloc_ref_count`)

wrapping struct `ArcData<T>`:

- use interior mutability of an optional
- keep extra info of shared counters referencing its data

drop implementation of weak pointer and strong pointer:

- strong pointer count is 0 $\implies$ drop `T`
- weak pointer + strong pointer count dropping to 0 $\implies$ droppping `ArcData<T>`

Rust drop order:

- run `Drop::drop` on object
- drop the object's fields 1 by 1 recursively

cloning pointers:

- `Arc`: increment weak counter, increment strong counter
- `Weak`: increment weak counter

dereferencing:

- `Arc`: unconditionally dereference since existence of `Arc` implies underlying `T` is valid
- `Weak`: upgrade to strong pointer
  - atomic increment and compare swap on strong counter to give out access

  - upgrade and return an `Arc` to caller
  - if strong pointer counter is 0 $\implies$ data doesn't exist and abort operation

strong pointer access to mutate data: runtime conditional chheck to allow exclusive access to `T`

- check weak pointer counter is 0, strong pointer count is 1
- cast underlyding data and return `& mut T`; safe since `Arc` exists

convert from strong pointer to weak pointer (downgrade): call clone on weak pointer and return the weak pointer

possible optimization: use 1 atomic counter instead of 2 $\implies$

- if user is not using weak pointers then they don't have to pay the cost when cloning /dropping
- use `ManuallyDrop<T>` instead of `Option<T>` to save an extra state and use existence of `Arc<T>` to know if data is gone or not
- let 1 weak count represets all existing `Arc<T>`s $\iff$ 1 `Arc<T>` left, decrement 1 weak count associated with all of them
- downgrade and `get_mut` requires more change:
- `get_mut`
  - need to check 2 atomic counters
  - temporarily lock downgrade operation by use of a special value indicating locked state for weak counter; use compare-exchange on `alloc_ref_count` (weak pointer + strong pointer) variable to replace with special value if the condition applies
  - check if strong pointer count == 1 $\implies$ we have exclusive access to data, replace special value earlier to unlock it (weak pointer + strong pointer count) and return `& mut T`
- downgrade
  - check that special value for `alloc_ref_count` (weak pointer + strong point counter) is not present, otherwise loop
  - compare-exchange acquire with `get_mut` method on `alloc_ref_count` atomic variable: increment if success and this will make future `get_mut` fail until `Weak::drop` makes this `alloc_ref_count` go back to 1 via release memmory ordering

6.4 Locks

7 Additional Ideas