Gathered notes from:

- Rust Atomics and Locks by Mara Bos [1]

1 Basic Concurrency Primitives Overview

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 fields of structs

un-implemented trait (`!Trait`) in some types:

```
Cell<T>: Send + !Sync where T: Send
* 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 consider for fairness of access

1.5 Park/Unpark

park: current thread put itself to sleep
unpark: another thread wakes sleeping thread; needs handle of the sleep thread 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)

1.6 Condition Variable

signaling events related to protected data of mutex

methods: `wait`, `notify_*`

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

1.6.1 Communication

waiting thread:

1. takes MutexGuard as input
2. unlocks mutex
3. thread put to sleep
4. thread wakes (via a notify of CondVar by another thread or spurious wakeup)
5. relocks mutex and returns MutexGuard

notifying thread:

1. invoke notify on CondVar

1.6.2 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

1.7 Comparison of Interior Mutability Primitives

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

1.8 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

1.9 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`

1.10 Traits for Shared Ownership Primitives

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

1.11 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

operations:

- `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
- mutability 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 invoke `drop` when no owners left
  eg: `std::rc::Rc`: clone increments counter only and gives reference to allocation
  eg: `std::sync::Arc`: version of `Rc` that is threadsafe

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 exist
- 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

atomic operation compiles to machine instructions

memory ordering at lowest level of individual instructions

tool for assembly code: `cargo-show-asm`

instructions that cannot be represented by 1 processor instruction, use equivalent implementation with composite/multiple instructions (eg: cmpxchg and loop)

4.1 x86-64

compare-exchange and compare-exchange-weak have no difference: compile to lock cmpxchg instruction

x86 lock prefix as a modifier for instructions:
`add, sub, and, not, or, xor, xchg(implicit), xadd, bts, btr, btc`

4.2 RISC

4.2.1 Load-Linked/Store-Condition (LL/SC)

used in a pair on a memory address

LL: returns current value of a memory address, processor remembers the address internally

SC: conditionally store new value to that memory address after previous LL if no updates occured since the LL

4.2.2 ARM64 LL/SC Pair

- `ldxr` (load exclusive register)
- `stxr` (store exclusive register)
- `clrex` (clear exclusive): stop tracking writes to memory
  $\Longrightarrow$ subsequent store conditional will fail

typically used in a loop with a branching compare to retry in order to ensure LL/SC succeeds

for efficient implementation:

- false negative can happen during store-conditional (eg: `stxr`): a chunk of memory tracked usually 64 bytes / 1 kB
- 1 memory address pre core can be tracked at a time

4.2.3 ARMv8.1 Atomic Instructions

- `cas` (equivalent to compare-exchange)
- `fetch-*` instructions

4.3 Cache Coherence Protocol for Consistency Between Processor Caches

eg: EMSI, MOESI, MESIF

keep states for individual cache levels

use `std::hint::black_box(..)` to disable compiler optimization when doing performance measurement

measurable difference when writes to cache require exclusive access at the same time when other cores are tring to load on the same cache lines

add padding to separate variables in cache lines (to reduce false sharing)
eg: `#[repr(align(64))]` for 64 byte alignment (note: must be power of 2, cannot be mixed with packed repr)

pack variable close together if they are expected to be accessed close temporally

out of order instructions/effects:

- store buffers for writing back to cache: brief moment of inconsistency present when write ops are not yet visible to other cores
- invalidation queues: invalidation requests (dropping cache line) queued for later processing
  $\implies$ inconsistency (outdated before they are dropped)
  $\implies$ visibility of write ops from other cores slightly delayed
- pipelining: parallel instructions: computation on some memory finishes before preceding instructions $\implies$ interaction with other cores (may appear out of order)
- special instructions exist to prevent these above

4.4 Memory Ordering

x86-64 and ARM64 are other-multi-copy atomic architectures $\implies$ write op visible to any core $\implies$ visible to all cores at same time
$\implies$ memory ordering is same as instruction reordering

x864-64 has fairly strong ordering restructions:

- release and acquire semantics have same cost as relaxed memory ordering
- store doesn't get reordered earlier than a preceding memory operation
- load doesn't get reordered later than a following memory operation
- $\implies$ (acquire $\iff$ relaxed, release $\iff$ relaxed)

ARM64 has relatively weak ordering: all memory ops can be reordered

- acquire and release versions of loads and stores:
  - `stlr` (store-release register)
  - `ldar` (load-acquire register)
  - `stlxr`(store-release exclusive register)
  - `ldaxr` (load-acquire exclusive register)
- none of the special acquire and release instructions is reordered with any other of these special instructions $\iff$ acquire-release operations are same as sequentially consistent operations (Acquire / Release / AcqRel has same cost as SeqCst)

4.5 Memory Fence / Barrier Instructions

prevents certain instructions being reordered past it

fences pose stronger constraint than release-acquire operations directly on atomic variables $\implies$ release-acquire atomic ops can be logically replaced with fences but the reverse is not true

fences are not acquire or load operations

Release Fence:

- usually used with atomic store after the release fence
- 1 way fence: memory operations before release fence cannot be reordered down past ALL subsequent memory writes after the fence (contrast with release store on atomic variable: memory operations before release op cannot be reordered down past the release op itself)

Acquire Fence:

- usually used with atomic load before the acquire fence
- 1 way fence: memory operations after acquire fence cannot be reordered up before ALL previous memory reads before the fence
- happens-before relation is established when load of the atomic variable reads any expected value that is caused by side-effects of the release sequence (eg: a release store on an atomic variable, or release fence followed by relaxed atomic store)

SeqCst Fence:

- both an release and an acquire fence

on X64-64:

- Acquire and Release are same as relaxed memory ordering (acquire/release fences are elided)
- SeqCst: issue of mfence instruction

on ARM64:

- Acquire: `dmb ishld`
- Release: `dmb ish`
- AcqRel: `dmb ish`
- SeqCst: `dmb ish` (same cost as acquire and release)

5 OS Primitives

futex used as a basic primitive that is optimized to avoid frequent syscall

originally syscall in linux but available in supporting libraries on other OSes (can use syscall from libc crate)

use of an atomic variable for thread notifications and wakeups

fast path in userspace when non-blocking, resort to slower syscall when blocking is required

spurious thread wakeups possible, thus need a condition check

solution to missing wakeup:

- use of another value:
  expected value match atomic variable $\implies$ blocking wait
  otherwise $\implies$ non-blocking
- wake is atomic with respect to wait:
  change atomic variable's value before wake
  $\implies$ thread that is about to wait will actually not block and skip wait therefore the wake up no longer has any effect
- check of expected value and wait/block (of `futex_wait`) happens as a single atomic operation wrt. other futex operations

priority inversion problem:

- high priority thread blocked by another lower priority thread with lock held
- solution: allow priority inheritance temporarily when this situation occurs
- see `FUTEX_<OP>_PI` operations

5.1 Implementation Variants on Other OS

Windows:

- heavy weight objects
- `critical_section`
- slim reader-writer lock
- address based waiting (`Wait/WakeByAddress`)

MacOS:

- `libc, libc++, obj-c, swift` interface: pthread impl
- platform specific lightweight `os_unfair_lock`, limitations: no cond var, no reader-writer variant

5.2 Standards for Accessing Kernel Scheduler via Special Libs or Syscalls

POSIX for Unix based systems

pthreads extensions: provide support for threading, data types, functions for concurrency

5.2.1 Concerns wrt. Movable/Non-Movable Types

pthread structures are generally non-movable types due to self references
possible workarounds:
`std::pin`
`Box<..>`: issue with leaking/forgetting an object: `pthread_mutex_destroy` on locked mutex may result in undefined behaviour as per spec.

5.3 Futex as an Efficient Mutex

simple futex-like addition to `C++` standard:
`std::atomic_wait`
`std::atomic_notify`

originally added to Linux systems: `SYS_futex` syscall: use of 32 bit atomic variable address to notify threads when to wake up

solution for missing wakeup signal: atomic op for wait $\implies$ a wake, between check of expect value provided to wait and the moment it goes to sleep, is not missed

manage state in userspace if possible, only rely on slower code path (via syscall) when absolutely necessary (need for a block)

usually wait used in a loop to check condition of possible spurious wakeups

futex related op arguments:

- 32 bit atomic pointer
- op constant
- optional flags, eg: `FUTEX_PRIVATE, FUTEX_CLOCK_REALTIME`
- remaining arguments dependent on the op

futex ops:

- `FUTEX_WAIT`: check and block is atomic
- `FUTEX_WAKE`: provide max number of threads to wake up
- `FUTEX_WAIT_BITSET`: wake only for bits set in common from a corresponding `FUTEX_WAKE_BITSET` op
- `FUTEX_WAKE_BITSET`
- `FUTEX_REQUEUE`
- `FUTEX_CMP_REQUEUE`
- `FUTEX_WAKE_OP`
- `FUTEX_PRIVATE_FLAG`

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 manage 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 one 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 (also 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 (when calling `split`): after 1st borrow expires, subsequent 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` from traits: `Deref`, `DerefMut`

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

requires `T: Send + Sync` and gives wrapping `Arc<T>` `Send + Sync`

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

cloning corresponds to incrementing counter and giving a shared reference to underlying data:

```
impl<T> Clone for Arc<T> {
  fn clone(&self) -> Self {
    self.data().ref_count.fetch_add(1, Relaxed);
    Arc {
      ptr: self.ptr,
    }
  }
}
```

only final decrement needs to be acquire and release, while all others can be only release:

```
impl<T> Drop for Arc<T> {
  fn drop(&mut self) {
  if var.fetchsub(1, release) == 1 {
    fence(acquire); //conditional acquire
    //drop logic
    unsafe {
      drop(Box::from_raw(self.ptr.as_ptr()));
    }
  }
}
```

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

```
 pub fn get_mut(arc: &mut Self) ->
  Option<&mut T> {
  if arc.data().ref_count.load(Relaxed) == 1 {
      fence(Acquire); //contional acquire
      //gained exclusive access now
      unsafe { Some(&mut arc.ptr.as_mut().data) }
  } else {
      None
  }
}
```

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 check 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

atomic-wait crate for providing cross platform interface for futex-like syscall:

- `wait(& AtomicU32, u32)`
- `wake_one(& AtomicU32)`
- `wake_all(& AtomicU32)`

platform implementation:

- Linux: `futex` syscall
- Windows: `WaitOnAddress`
- FreeBSD: `_umtx_op`
- MacOS: `libc++`

6.4.1 Constructing Mutex

use atomic variable for tutex-like syscall (invoke wait for locking):

```
struct Mutex<T> {
  state:: AtomicU32,
  value: UnsafeCell<T>,
}

struct MutexGuard<'a, T> {
  mutex: & 'a Mutex<T>,
}

//Produce a guard as proof of exclusive access.
//
//add Deref and DerefMut traits for data access
//like & T, & mut T
fn lock(& self) -> MutexGuard<T> {
  while self.state.swap(1, Acquire) == 1 {
    wait(&self.state, 1); //wait until state != 1
  }
  MutexGuard { mutex: self }
}
```

use drop of guard to unlock mutex by invoking wake after setting state to unlocked value:

```
imnpl<T> Drop for MutexGuard<'_, T> {
  fn drop(& mut self){
    self.mutex.state.store(0, Release);
    // sufficient for 1 thread to claim lock
    wake_one(&self.mutex.state);
  }
}
```

note: removal of wait and awake pair (atomic operations) $\implies$

- still correct wrt. memory safety
- equivalent to spin lock
- serve as optimization

helper crate (`lock_api`) to generate API/boilerplate for Mutex related things where user provides:

- type representing lock state
- unlock and lock functions in `lock_api::RawMutex` trait

futher optimizations to avoid syscall:

- avoid unconditional awake when dropping MutexGuard: use extra info to track if there are no other threads waiting:
  - split state into more values:
    - 0: unlocked
    - 1: locked, no threads waiting
    - 2: locked, ≥ 1 threads waiting
  - uncontended case: `wait` and `wake_one` are both avoided
  - contended case: thread that waits will need to eventually do `wake_one`
- further optimize by incorporating spin wait for brief duration before resorting a heavy cost wait:
  - use case when exclusive access is needed only for a very short time
  - insert spin loop for a number of iterations for the case when there are no other waiters (state value of 1)

6.4.2 Constructing Condition Variable

for use with a lock (eg: mutex)

unlocks mutex on `wait()`

locks on a notify signal from another thread

checks on some supplied condition to enable further access to critital section for associated thread

thread may be spuriously woken up which is kept in check by the supplied condition, however this lcoks and unlocks mutex so it takes up compute cycles

mechanism to prevent lost signal is different to futex:

- cond var starts to listen to signal before unlocking mutex
- futex uses a check of state of atomic variable to make sure waiting is a good idea

possible implementation of cond var using futex:

- let every notification change an atomic variable
- `wait()` load the atomic variable before unlocking and call futex wait with the said atomic variable (and let futex wait return before return on a received notify signal)
  $\implies$ cond var's most simplistic impl. only needs 1 atomic variable and futex calls, `wait` and `wake/notify`
- spurious wakeups on waiting thread: wrap an outer loop around `wait()` call in order to check for user supplied condition

```
fn wake(){
  atomic_var++;
  futex::wake(atomic_var);
}

fn wait(guard: MutexGuard) -> MutexGuard {
  //already locked
  let counter = atomic_var.load(Relaxed);
```

```
    let mtx = guard.mutex;

    //explicit drop does a release store on some
    //atomic variable associated with the mutex
    //
    //this happens-before another thread locking
    //mutex (acquire load on some atomic variable
    //associated with the mutex) and signaling,
    //therefore relaxed load of atomic variable
    //by wait() is enough
    guard.drop();

    //unlocked now

    futex::wait(atomic_var, counter);

    mtx.lock() //lock again
  }
```

futex wake-wait pair is atomic $\iff$ either waiting thread goes to sleep and get woken up later or the thread does not go to sleep and continues on

optimization:

- for the case of wait: not much optimization since thread has decided to call wait anyway after checking supplied condition
- for the case of wake: avoid call if no waiting threads present:
  introduce variable to track number of waiters (increment when prior to waiting, decrement when done waiting), notify threads can skip if count is 0
- optimization to reduce spurious wakeups: multi-group of waiters and swapping
- optimization to reduce `notify_all`'s thundering herd problem: use futex requeue

6.4.3 Constructing Reader-Writer Lock

`T: Send + Sync` $\implies$ `RWLock<T>: Send + Sync`

`T` requires `Send` just like the case for `Mutex<T>`

`T` additionally requires `Sync` because it may be shared between threads for multiple readers

idea: counter for shared readers, cell for interior mutability

2 lock types to access data:

- `ReadGuard`: allows ≥ 1 readers at the same time
- `WriteGuard`: behaves similar to `Mutex` (exclusive access from 1 thread at a time

optimizations:

- introduce another counter variable for waiting writers in order to reduce writer spin looping and waking up excessively due to presence of readers
- writer starvation problem: add additional states to account for cases of waiting writer:
  when writer lock not acquired $\implies$ 2 * (# of readers) + $\mathbb{1}$(any waiting writer present)
  when writer lock acquired $\implies$ `U32::Max`

  (writer present $\iff$ odd number) $\implies$ readers need to be blocked

6.4.4 summary

- `atomic-wait` crate for futex-like interface on major OSes
- efficient mutex implementation tracks info on waiting threads to avoid extra syscalls
- cond var may track number of waiting threads to avoid extra wake operations
- additional variables are used in lock to wake writers independently from readers
- extra state may be used to prioritize waiting writer over waiting readers

7 Additional Ideas

todo

References

[1] Mara Bos. Rust atomics and locks, 2015.