CS3211 Useful Stuff from Tutorials and Assignments

by Devansh Shah

C++

Tutorial 0.5 : Introduction to C++

- Upon scope exit (e.g. function returns, or end of curly braces), the destructors of each local variable are called in the reverse order that they were constructed. This is why we don't have to explicitly call any `new` or `delete`.
- Variables are generally word-aligned (depending on machine). Example:

```
struct X {
  char a;
  int b; // 3 bytes of padding before b
};
```

- By default, all variables are passed by value in C++. We can change this behaviour by adding an ampersand to the parameter list in the function definition.

```
void demo() {
  B a; // Default constructor, no arguments
  B a_copy = a; // Copy constructor, argument is another instance of B
  B b(1); // Custom constructor, int as argument
  b = a; // Copy assignment operator, RHS is another instance of B
}
```

- When a class only defines a move constructor / assignment operator but NOT a copy constructor / assignment operator, it can only be moved and not copied. This is what types like `std::thread` and `std::unique_ptr` do. When using such types, it's important to ensure ownership is properly transferred, e.g. using `std::move`.

Tutorial 1 : Threads and Synchronization

- Trying to access a variable that has already been destructed/deallocated will is called Use-After-Free (UAF), and is undefined behaviour in C++.

```
std::string* raw_world_ptr = nullptr;
{
    std::string world{"12"};
    raw_world_ptr = &world;
}
std::cout << *raw_world_ptr << std::endl; // UAF - undefined behaviour!
```

- A thread that has finished executing code, but has not yet been joined is still considered an active thread of execution and is therefore joinable. If you call a destructor on a joinable thread (e.g. the main program exits without calling `.join()`, then `std::terminate()` is called and the program terminates with a non-zero return value. However, this is not UB.

```
void test() {
    std::cout << "hello!\n";
}
int main() {
    std::thread t1(test);
    return 0; // "Program terminated with SIGSEGV"
}
```

- Use-After-Move is unspecified behaviour, not undefined behaviour. The following code is "valid" but returns a non-zero value (i.e., error):

```
int main() {
    std::thread t1(test);
    std::thread t2 = std::move(t1);
    t1.join();  // what(): invalid argument; SIGSEGV
    return 0;
}
```

- You cannot `.join()` the same thread twice Because a post-condition of `.join()` is that `.joinable()` is `false`, and the second `.join()` will throw an error.
- Binary semaphore $\neq$ Mutex. They're used for different purposes. Mutexes are used to implement critical section, whereas semaphores are used as a signalling mechanism. You can `acquire` a semaphore on one thread, and `release` on another, but you cannot do this using mutexes.
- Condition variables are prone to spurious wakes - they may acquire the lock and "wake up" even when the condition is `false`.

```
while (jobs.empty()) {
    cond.wait(lock);
}
// equivalent code:
cond.wait(lock, [this]() {returns !jobs.empty(); });
```

Tutorial 2 : Atomics in C++

- Atomics are processor-level constructs i.e. different (special) instructions are generated. Atomics work faster than mutexes and semaphores because these are implemented using atomics under the hood, and so, they incur additional overhead.

Tutorial 3 : Debugging Concurrent C++ Programs

- "The modern C++ memory model exhibits a different behavior when it is running on a weak-consistent architecture (such as ARM)." - False. We should not need to care about the exact hardware when programming! C++ compilers guarantee that the behaviour of our program is the same (i.e., adheres to the C++ specifications). It does this by inserting special instructions (which leads to more overhead) when compiling programs for a weak-consistent architecture.
- Debugging tools cannot find all bugs. Having no errors does not mean your code is correct.
- `std::shared_ptr` is **not thread safe**. The increment/decrement of the reference count is atomic (and hence, thread-safe) but accesses to the underlying object itself are not thread-safe. So, we still need to use synchronization while performing conflicting actions on the object (which `shared_ptr` tracks.

- `std::jthread` is like `std::thread`, only without the stupid. `std::thread`'s destructor would terminate the program if you didn't join or detach it manually beforehand. This led to tons of bugs, as people would expect it to join on destruction. `jthread` fixes this; it joins on destruction by default (hence the name: "joining thread").
- If you pass shared pointers by reference, then you can have a data race on the shared pointer itself (since they refer to the same object). Example:

```
int main() {
    std::shared_ptr<int> ptr;
    auto reader = std::jthread([](std::shared_ptr<int>& ptr) {
        while(ptr == nullptr); // data race!
        printf("%d\n", *ptr); }, std::ref(ptr));
    auto writer = std::jthread([](std::shared_ptr<int>& ptr) {
        for(int i = 0; i < 100; i++)
            ptr = std::make_shared<int>(i); // data race!
    }, std::ref(ptr));
}
```

  Solution: never pass shared pointers by reference!
- Notice in the above code, we had to explicitly pass a reference of the shared pointer, using `std::ref`, to the thread. In regular functions, we don't have to do this - we can simply add an `&` in the parameter of the function header. But for threads, it is not sufficient for the thread's entry-point function to take a reference type: the thread object itself takes its arguments by value. This is because you usually want a copy of objects in a separate thread. To get around this, you may pass `std::ref`, which is a reference wrapper hiding reference semantics under a copyable object.
- It's possible for reference counting to fail to delete the objects when there are **circular references**. For example, if a thread creates a Doubly Linked List and then exits, the nodes in the DLL have a positive reference count because they point to each other (even though the programmer can never access these objects anymore). This leads to **memory leaks**.
- Here's an implementation of a shared pointer using atomics. notice that we use `std::memory_order_ac_rel` for the destructor and use `std::memory_order_relaxed` for the constructor, since we don't have to synchronise the constructors, only the destructors):

```
template <typename T>
class SharedPtr {
    std::atomic<size_t>* m_count; T* m_ptr;
public:
    SharedPtr(T* ptr) : m_count(new std::atomic<size_t>(1)),
                        m_ptr(ptr) {}
    SharedPtr(const SharedPtr& other) : m_count(other.m_count),
        m_ptr(other.m_ptr) {
        m_count->fetch_add(1, std::memory_order_acq_rel); }
    ~SharedPtr() {
        size_t old_count = m_count->fetch_sub(1,
                            std::memory_order_acq_rel);
        if(old_count == 1) {
            delete m_ptr; delete m_count;
        }
    }
};
```

- Rule of thumb: use `acq_rel` when you don't need a single total modification order. Use `relaxed` when you don't require any synchronisation at all - typically when you require atomicity but not synchronisation.

Tutorial 5 : Lock-Free Data Structures

- Compare-And-Swap (CAS): `variable.compare_exchange(old, new)` will change the value of `variable` from `old` to `new` atomically, i.e., if the current value of `variable` is not `old`, it does nothing.
- The weak version of `compare_exchange_weak` can fail spuriously. The strong version guarantees no spurious failures.
- **ABA Problem**: When using compare-and-swap loops, one thing that must be immediately addressed is the ABA problem.
  We made the assumption that because we perform steps A+B+C atomically with a CAS loop, we only succeed if the value of `m_queue_front` was still `old_front` (the expected value) at the time of the write.
  But if `old_front` can be set to another value, and then set back to the same value again in time for the CAS operation to be performed, the CAS would succeed! This is because previously freed memory addresses may have been allocated to subsequent calls to `new Node()`.
  This is where the name comes from: a value initially has value A, is set to B, and then back to A.
- Basically, the main reason that we have the ABA problem is because the pointer value of `old_front` is the same as the pointer value of `m_queue_front`, i.e., the pointers hold the same address — even though the identity of the node has changed!
- We can solve the ABA problem using **generation counters**. The idea is that we tie a unique number together with the pointer, so that even if the address is the same, the value (which is a combination of both the pointer and the counter) is different.

```
struct alignas(16) GenNodePtr
{
    Node* node;
    uintptr_t gen;
};
```

  So now, for us to get a false positive on the CAS check, we must have had both of the following conditions be true:
  - a new `m_queue_front` was allocated at the same address as our old one.
  - $2^6$4 pops have happened (causing the conter to overflow and wrap around) while our consumer was asleep

  This is highly unlikely, mainly due to the second condition.
- There are 2 ways we can check whether any object is lock-free: the `is_lock_free` method on instance, and the static data member `is_always_lock_free`. The reason there's two is because a given object might only be atomic if aligned suitably, and the alignment can be runtime-dependent. If we want to know whether a type is always

lock free, then we can use `std::atomic<T>::is_always_lock_free` — this is true if the type is always lock-free, regardless of its alignment.

- Every complete object type has a property called **alignment requirement**, which is an integer value of type `size_t` representing the number of bytes between successive addresses at which objects of this type can be allocated. The valid alignment values are non-negative integral powers of two.
- One solution to Use-After-Free (UAF) concerns is to just never free anything explicitly. Just mark the nodes for deletion, and when the last thread leaves, we free all of them at once. Another solution would be to use reference counting (i.e., an atomic `shared_ptr` to know when there are no more remaining references to a particular object.
- For a lock-free data structure to be "correct", we require **linearizability**. That is, all operations on the data structure must be totally ordered, consistent with happens-before and the results of the queue. The fundamental problem with our queue is that it is possible for a thread to read values while another thread is performing its own operation - during this time, the state of the queue is inconsistent and our invariants are broken (this is generally okay as long as no one else is able to see this inconsistent state). So, we read values which seem to violate our requirements for a "correct" queue.

Tutorial 7 : Classical Synchronization Problems in C++ and Go

- **H2O problem** - solution using semaphores and barrier.

```
// To prevent illegal bonding, only reset the semaphores after we bond.
// The barrier will only allow a new batch of atoms through if the
// semaphores are fully reset, and this only happens after all 3 atoms
// have bonded.
struct WaterFactory3 {
  std::counting_semaphore<> oxygenSem;
  std::counting_semaphore<> hydrogenSem;
  std::barrier<> barrier;

  WaterFactory3() : oxygenSem{1}, hydrogenSem{2}, barrier{3} {}

  void oxygen(void (*bond)()) {
    oxygenSem.acquire();  // Lets at most one oxygen through
    barrier.arrive_and_wait();
    bond();
    oxygenSem.release();  // We are done, let the next oxygen in
  }

  void hydrogen(void (*bond)()) {
    hydrogenSem.acquire();  // Lets at most two hydrogen through
    barrier.arrive_and_wait();
    bond();
    hydrogenSem.release();  // We are done, let the next hydrogen in
  }
};
```

- Note that barriers are reusable, whereas latches are not. So, if we replace `barrier` with `latch` in the above code, we would only be able to generate one valid H2O molecule.
- **FIFO Semaphore** - We can implement a FIFO semaphore using an idea similar to how a ticketing system works. Each thread gets a queue number, and it is allowed to proceed only when all threads with a smaller queue number have already passed through. Here, we can use relaxed consistency while obtaining our queue number and still be sure that it is unique (because even though they don't partake in synchronizes-with relationships, atomics still guarantee a total MO for that particular variable, which all threads agree on).

```
struct FIFOSemaphore {
  std::mutex mut;
  std::condition_variable cond;
  std::atomic<std::ptrdiff_t> next_ticket;
  std::ptrdiff_t now_serving;

  FIFOSemaphore(std::ptrdiff_t initial_count)
      : mut{}, cond{}, next_ticket{1}, now_serving{initial_count} {}

  void acquire() {
    std::ptrdiff_t my_ticket =
        next_ticket.fetch_add(1, std::memory_order_relaxed);
    std::unique_lock lock{mut};
    cond.wait(lock, [=]() { return now_serving >= my_ticket; });
  }

  void release() {
    {
      std::scoped_lock lock{mut};
      now_serving++;
    }
    cond.notify_all();
  }
};
```

We can even replace the condition variables with atomic waits (introduced in C++ 20), like so:

```
struct FIFOSemaphore {
  std::atomic<std::ptrdiff_t> next_ticket;
  std::atomic<std::ptrdiff_t> now_serving;

  FIFOSemaphore(std::ptrdiff_t initial_count)
      : next_ticket{1}, now_serving{initial_count} {}

  void acquire() {
    auto my_ticket = next_ticket.fetch_add(1,
                                           std::memory_order_relaxed);
```



```
    auto my_out_ticket = now_serving.load(std::memory_order_acquire);
    while (my_out_ticket < my_ticket) {
      // Just like a cond var, wait can spuriously wake up.
      // Before, mutexes were used to protect the cond var
      // to ensure that between checking the cond var
      // and waiting, the underlying data structure
      // is not changed beneath our feet.
      // With an atomic wait, this is replaced with an `old` parameter,
      // and the wait will only block if now_serving was still equal to
      // the old value.
      // Since our waiting condition can only change between true and
      // false when now_serving is no longer my_out_ticket, this
      // application of atomic wait is correct.
      now_serving.wait(my_out_ticket, std::memory_order_relaxed);
      my_out_ticket = now_serving.load(std::memory_order_acquire);
    }
  }

  void release() {
    now_serving.fetch_add(1, std::memory_order_acq_rel);
    now_serving.notify_all();
  }
};
```

Another way to implement FIFO semaphore is to use an explicit queue of semaphores (one for each thread), and a mutex to protect the shared queue (to prevent data races), as follows:

```
struct FIFOSemaphore {
  struct Waiter {
    std::binary_semaphore sem{0};
  };

  std::mutex mut;
  std::queue<std::shared_ptr<Waiter>> waiters;
  std::ptrdiff_t count;

  FIFOSemaphore(std::ptrdiff_t initial_count)
      : mut{}, waiters{}, count{initial_count} {}

  void acquire() {
    auto waiter = std::make_shared<Waiter>();
    {
      std::scoped_lock lock{mut};
      if (count > 0) {
        count--;  // Positive count,
        return;   // simply decrement without blocking
      }
      waiters.push(waiter);  // Zero count, add to waiters
    }
    waiter->sem.acquire();  // and block on the semaphore
  }

  void release() {
    std::shared_ptr<Waiter> waiter;
    {
      std::scoped_lock lock{mut};
      if (waiters.empty()) {
        count++;  // No waiters, simply increment count
        return;
      }

      waiter = waiters.front();  // Pop a waiter
      waiters.pop();
    }
    waiter->sem.release();  // and signal it
  }
};
```

- **Barber Problem** - A barbershop consists of a waiting room with n chairs and the barber chair. If there are no customers to be served, the barber goes to sleep. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop.

```
// Initialization
customers = 0
mutex = Semaphore (1)
customer = Semaphore (0)
barber = Semaphore (0)
customerDone = Semaphore (0)
barberDone = Semaphore (0)

// Customer Pesudo-Code
wait(mutex);
if (customers == n) {
    signal(mutex);
    exit();
}
customers += 1;
signal(mutex);
signal(customer);
wait(barber);
getHairCut ();
signal(customerDone);
wait (barberDone);
```

```
wait(mutex);
customers -= 1;
signal(mutex);

// Barber Pseudo-Code
while (TRUE) {
    wait(customer);
    signal(barber);
    cutHair();
    wait(customerDone);
    signal(barberDone);
}
```

Golang

Tutorial 5 : Goroutines and Channels

- When you call `defer`, that statement is run *just before* the function exits. In particular, it is run *after* all the other statements in the function are executed.
- The reason **the following code deadlocks** is because the last goroutine to write into the channel blocks - since the main goroutine needs to wait until `wg.Done()` is called by that goroutine, while that goroutine is waiting for main to read from the channel (which is not possible because main is blocked at `wg.Wait()`):

```go
func main() {
    ch := make(chan int) // make "unbuffered" channel
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            count := <-ch // blocking dequeue
            count++       // safely add 1 as the exclusive owner
            ch <- count   // blocking enqueue (for another consumer)
        }()
    }
    ch <- 0 // main sends initial value; blocking enqueue
    wg.Wait()                       // wait for all goroutines
    fmt.Println("Count: ", <-ch) // dequeue final result
}
```

  The fix is to move `wg.Done()` to be just before `ch <- count`, or to use a buffered channel (of any size).
- **Producer-Consumer paradigm** using channels:

```go
func producer(done chan struct{}, q chan<- int) {
    for {
        select {
        case q <- 1: // keeps incrementing...
        case <-done: // until stopped (channel closed)
            return
        }
    }
}

func consumer(done chan struct{}, q <-chan int, sumCh chan int) {
    sum := 0
    for {
        select {
        case val := <-q:
            sum += val
        case <-done:
            sumCh <- sum
            return
        }
    }
}

var (
    NumProducer = 5
    NumConsumer = 5
)

func main() {
    done := make(chan struct{})
    q := make(chan int)
    sumCh := make(chan int, NumConsumer)

    for i := 0; i < NumProducer; i++ {
        go producer(done, q)
    }

    for i := 0; i < NumConsumer; i++ {
        go consumer(done, q, sumCh)
    }

    time.Sleep(time.Second) // run for 1 second
    close(done)             // stop all producers and consumers

    sum := 0
    for i := 0; i < NumConsumer; i++ {
        sum += <-sumCh
    }
    fmt.Println("Sum: ", sum)
}
```

  We could also have different `sumCh` for each consumer to send back its local sum to the main goroutine. This increases concurrency since it prevents "head of line blocking" (even if the first consumer is taking very long to send its local sum, we can get the sums from other consumers in the meantime). So, we have:

```go
for _, ch := range sumChs {
    sum += <-ch
}
```

- **Channels are not lock-free**. `chan`'s internal implementation uses a lock. So, if a goroutine holds the channel's mutex and gets suspended, nobody else can use the channel!
- By default channels are like blocking queues. We can implement non-blocking queue by using `select-default` like so (say, for `try_enqueue`):

```go
func (q *queue) tryEnqueue(num int) bool {
    select {
    case q <- num:
        return true
    default:
    }
    return false
}
```

- We can use contexts to send signals to the goroutines. These avoids us having to use a separate `done` channel. Contexts can have timeouts, and we can also manually call`cancel`. The key idea is that the goroutines don't need to know why they've been cancelled.

```go
func main() {
    c := make(chan int)
    ctx, cancel := context.WithTimeout(context.Background(),
                1 * time.Second)

    go func() {
        for {
            c <- 9
            time.Sleep(200 * time.Millisecond)
        }
    }()

    done := make(chan struct{})
    go func() {
        for {
            select {
                case s := <- c:
                    fmt.Println(s)
                case <- ctx.Done():
                    fmt.Println("I've been cancelled")
                    close(done)
                    return
            }
        }
    }()

    // cancel()
    <- done
}
```

Tutorial 6 : Concurrency Patterns in Go

- Fan-out Fan-in pattern is like demultiplexing-multiplexing. We distribute (aka "fan out") tasks to multiple goroutines and then combine/pipe/multiplex (aka "fan in") the output into a single goroutine via a channel.
- The number of go routines spawned matters. In general, create `runtime.NumCPU()` goroutines or profile your code to enhance performance.

- To manage different exit conditions, we can create context trees. Suppose the `main` goroutine must exit after 4 seconds, goroutine 2 must stop after 2 seconds or when `main` exits, and goroutine 3 must stop if the program receives a termination signal or when `main` exits. Then, we could do this:

```go
var (
    timeout1 = 4 * time.Second
    timeout2 = 2 * time.Second
)

func main() {
    startTime := time.Now()
    ctx, _ := context.WithTimeout(context.Background(), timeout1)

    ctx2, _ := context.WithTimeout(ctx, timeout2)
    go func() {
        <-ctx2.Done()
        fmt.Printf("ctx2 done at %v\n",  time.Now().Sub(startTime))
    }()

    ctx3, cancel := context.WithCancel(ctx)
    go func() {
        <-ctx3.Done()
        fmt.Printf("ctx3 done at %v\n", time.Now().Sub(startTime))
    }()

    go func() {
        <-handleSigs()
        cancel()
        fmt.Printf("signal in at %v\n", time.Now().Sub(startTime))
    }()

        <-ctx.Done()
    fmt.Printf("ctx done at %v\n", time.Now().Sub(startTime))
}
```

- If you want to print the events in a particular order (say, in increasing order of task id) after fanning-in through an output channel, you can use a serializer which buffers out-of-order results (exactly how out-of-order packets are buffered by the receiver in networking). Example:

```go
// Serializer goroutine
finalOutputCh := make(chan Event, 1)
go func() {
    eventMap := make(map[int64]Event)
    var curEventId int64 = 1
    for output := range outputCh {
        // is this the next event we're looking for?
        if output.id == curEventId {
            // Yes, then push it to the final reader
            finalOutputCh <- output
            curEventId += 1
            // check if there's a cascade of events to let go
            for {
                if event, present := eventMap[curEventId]; present {
                    finalOutputCh <- event
                    curEventId = curEventId + 1
                } else {
                    break
                }
            }
        } else {
            // buffer out-of-order events
            eventMap[output.id] = output
        }
    }
    close(finalOutputCh)
}()
```

- In case of a pipelined approach, if each stage takes a different amount of time, and you have a limited number of goroutines, allocate resources (goroutines) proportional to how much time each stage task - this ensures that the load at every stage is nearly evenly distributed.

Tutorial 7 : Classical Synchronization Problems in C++ and Go

- **H2O problem** - Oxygen leader solution. Oxygen takes charge and acts as the leader. We can think of it as 3 main steps - pre-commit, commit, and post-commit (inspired by how git works).
  1. Pre-commit: oxygen waits for hydrogens, and hydrogens inform oxygen that they've arrived
  2. Commit: 1 oxygen and two hydrogens call `bond`
  3. Post-commit: oxygen waits for hydrogens to finish executing `bond` too, then steps down as leader (to allow another oxygen to take over the leader role).

```go
type WaterFactoryWithLeader struct {
    oxygenMutex chan struct{}
    precomH     chan chan struct{}
    commit      chan chan struct{}
}

func NewFactoryWithLeader() WaterFactoryWithLeader {
    wf := WaterFactoryWithLeader{
        oxygenMutex: make(chan struct{}, 1),
        precomH:     make(chan chan struct{}),
        commit:      make(chan chan struct{}),
    }
    wf.oxygenMutex <- struct{}{}
    return wf
```



```go
}

func (wf *WaterFactoryWithLeader) hydrogen(bond func()) {
    commit := make(chan struct{}) // 1: Private communication channel
    wf.precomH <- commit          // 2: (Precommit)
    <-commit                      // 3: (Commit)
    bond()                        // 4: Bond
    commit <- struct{}{}          // 5: (Postcommit)
}

func (wf *WaterFactoryWithLeader) oxygen(bond func()) {
    // Step 1: Become leader
    <-wf.oxygenMutex // For fun, we can use a channel as a mutex

    // Step 2: (Precommit)
    //         Receive arrival requests from 2 hydrogen atoms
    h1 := <-wf.precomH
    h2 := <-wf.precomH

    // Step 3: (Commit)
    //         Tell the 2 hydrogen atoms to start bonding
    h1 <- struct{}{}
    h2 <- struct{}{}

    // Step 4: Bond
    bond()

    // Step 5: (Postcommit)
    //         Wait until the 2 hydrogen atoms to finish
    // We re-use the same communication channel as (Commit)
    <-h1
    <-h2

    // Step 6: Step down from being leader
    wf.oxygenMutex <- struct{}{}
}
```

- **FIFO Semaphore** - Can we use Go channels directly to implement a FIFO semaphore since channels guarantee FIFO ordering of the values sent across it? No! Even though the values themselves are sent FIFO through the channel, the order in which goroutines are allowed to send and receive values is not FIFO. In particular,
  - If item A goes into channel before item B, then item A will be popped out before item B → Guaranteed by standard
  - But if reader A blocks on channel before reader B, does not mean that A will be unblocked before B!

  But it just so happens that Go's (current) implementation of channels is indeed FIFO, but this is not guaranteed by the specification.

Anyway, we can implement a FIFO sempahore using a queue of (regular) FIFO - the idea is that each goroutine blocks on its own private sempahore (so we don't need to rely on the ordering semantics of the semaphore at all), and a daemon goroutine manages a queue of semaphores (on which the other goroutines are blocked) to unblock them in a FIFO manner, when other goroutines leave.

```go
type FIFOSemaphore struct {
    acquireCh chan chan struct{}
    releaseCh chan struct{}
}

func FIFOSemaphore(initial_count int) *FIFOSemaphore {
    sem := new(FIFOSemaphore)
    sem.acquireCh = make(chan chan struct{}, 100)
    sem.releaseCh = make(chan struct{}, 100)

    go func() {
        count := initial_count
        // The FIFO queue that stores the channels used to unblock waiter
        waiters := NewChanQueue()
        for {
            select {
            case <-sem.releaseCh: // Increment or unblock a waiter
                if waiters.Len() > 0 {
                    ch := waiters.Pop()
                    ch <- struct{}{} // Unblocks the oldest waiter
                } else {
                    count++
                }

            case ch := <-sem.acquireCh: // Decrement or add a waiter
                if count > 0 {
                    count--
                    ch <- struct{}{} // Don't keep waiter blocked
                } else {
                    waiters.PushBack(ch) // Add waiter to back of queue
                }
            }
        }
    }()

    return sem
}

func (s *FIFOSemaphore) Acquire() {
    ch := make(chan struct{})
    // Send daemon a channel that can be used to unblock us
```

```
    s.acquireCh <- ch
    // Block until daemon decides to unblock us
    <-ch
}

func (s *FIFOSemaphore) Release() {
    s.releaseCh <- struct{}{}
}
```

Rust

Rust Safety and Concurrency

- Ownership Rules:
  - Each value in Rust has an owner.
  - There can only be one owner at a time.
  - When the owner goes out of scope, the value will be dropped.
- You can do as many immutable borrows as you like at the same time! As long as nobody modifies the data, the correctness is maintained.
- `println!` is a macro and the `&` is inserted for you, i.e., `println!` also borrows objects (immutably), does not own them.

```
fn f(x: &mut Vec<u8>) {
    x[1] = 244;
    print!("Borrowed version of x in f: {:?}", x) // [1, 244, 3, 4]
}
fn main() {
    let mut x: Vec<u8> = vec![1, 2, 3, 4];
    f(&mut x);
    print!("x from end of main: {:?}", x); // [1, 244, 3, 4]
}
```

- Note that by default all variables are immutable (aka `const`). You need to explicitly specify `mut` to be able them.
- The following code doesn't compile because the compiler cannot be sure that the lifetime of the "borrowed" return will live as long as our inputs.

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

- We can use lifetimes to solve the above issue. Here, we're telling the compiler to find a lifetime `'a` such that `x` and `y` live at least as long as lifetime `'a`, and the return value also lives at least as long as lifetime `'a`. Now, the compiler can go and verify this as part of its type system!

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

In the above code, if we try to return a local variable `z` that is created within the `longest` function, the compiler will throw an error (because the function would return a dangling reference!)

- If we use `move` on primitives that have a Copy-trait, the variables are just copied. So, in the following code, each thread has a local variable called `counter` that they increment (no longer shared variable).

```
use std::thread;

fn main() {
    let mut counter = 0;
    let t0 = thread::spawn(move || { counter += 1; });
    let t1 = thread::spawn(move || { counter += 1; });
    t0.join();
    t1.join();
    println!("{}", counter);
}
```

- When we write a concurrent counter by wrapping the shared variable in a mutex, the compiler is still not convinced that our threads finish before the end of `main()`, when the counter is dropped. So, in addition to the mutex, we also have to use `Arc` (Atomically Reference Counted) - very similar to `shared_ptr` in C++. Now, the `counter` will only be dropped if both references die - and the compiler is satisfied.

```
use std::thread;
use std::sync::{Arc, Mutex};

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let counter1 = counter.clone();
    let counter2 = counter.clone();
    let t0 = thread::spawn(move || { *counter1.lock().unwrap() += 1 });
    let t1 = thread::spawn(move || { *counter2.lock().unwrap() += 1 });

    t0.join();
    t1.join();
```



```
    println!("{}", *counter.lock().unwrap());
}
```

- We can avoid using `Arc` if we just convince the compiler that our threads will complete running before the variables are dropped. This is where **scoped threads** come in. We can help the compiler understand our intention by using an automatically scoped thread - it automatically joins before the end of the scope (like `std::jthread` from C++). So, the following code works as expected:

```
use std::thread;
use std::sync::{Mutex};

fn main() {
    let counter = Mutex::new(0);
    thread::scope(|s| {
        s.spawn(|| *counter.lock().unwrap() += 1);
        s.spawn(|| *counter.lock().unwrap() += 1);
    });
    println!("{}", *counter.lock().unwrap());
}
```

- **Interior Mutability**: In the above code, two threads are mutably changing a non-mutable reference. Doesn't this seem crazy? Yeah, it's possible in Rust because the `Mutex` type contains an `UnsafeCell` to indicate that this code is "unsafe". Using "unsafe" disables the borrow-checker for that particular part.
- If we don't want to use mutexes either, we can use atomics, exactly as how we would in C++!

```
use std::sync::atomic::{AtomicI32, Ordering};
use std::thread;

fn main() {
    let counter = AtomicI32::new(0);
    thread::scope(|s| {
        s.spawn(|| {
            counter.fetch_add(1, Ordering::Relaxed);
        });
        s.spawn(|| {
            counter.fetch_add(1, Ordering::Relaxed);
        });
    });
    println!("{}", counter.load(Ordering::Relaxed));
}
```

- Finally, another way to convince the compiler that our variables will live as long as the threads do, is to use the `static` keyword - then we can do away with scoped threads. Now, the compiler can prove that `COUNTER` will last for the whole program.

```
use std::thread;
use std::sync::atomic::{AtomicI32, Ordering};

static COUNTER: AtomicI32 = AtomicI32::new(0);

fn main() {
    let t0 = thread::spawn(|| { COUNTER.fetch_add(1, Ordering::Relaxed); });
    let t1 = thread::spawn(|| { COUNTER.fetch_add(1, Ordering::Relaxed); });
    t0.join().unwrap();
    t1.join().unwrap();
    println!("{}", COUNTER.load(Ordering::Relaxed));
}
```

- Using **Rayon** makes it easy to data parallelism. Observe the use of `into_par_iter()` here:

```
use rayon::prelude::*;

fn magic_sum(from: u128, to: u128) -> u128 {
    (from..to).into_par_iter().filter(|i| i % 7 == i % 5).sum()
}

fn main() {
    let (from, to) = {
        let mut args = ["", "0", "100000000"].iter();
        args.next(); // skip argv[0]
        (args.next().unwrap(), args.next().unwrap())
    };
    println!("{}", magic_sum(from.parse().unwrap(), to.parse().unwrap()))
}
```

- When we have nested scoped threads, we need to be careful to `move` the value to the inside threads, like so:

```
use std::thread;

fn main() {
    let  arr = vec![String::new(); 10];
    thread::scope(|s| {
        // Note that &T is Copy - so a move copies this reference
        let borrowed_arr = &arr; // We can borrow this, and move i
        for i in 0..10 {
            // We can do this too!
            // let borrowed_arr = &arr;
            s.spawn(move || println!("{}", borrowed_arr[i]));
        }
    })
}
```

- Example of using `tokio` to write async code for a TCP server:

```
#[tokio::main]
async fn main() -> std::io::Result<()> {
```

```
    let port = std::env::args()
        .nth(1)
        .map(|s| s.parse().unwrap())
    let listener = TcpListener::bind(SocketAddr::from(([127, 0, 0, 1],
        port))).await?;
    loop {
        let (socket, _) = listener.accept.await?;
        tokio::spawn(async move {
            eprintln!("Accepted conn!");
            std::mem::drop(handle_client(socket).await);
            eprintln!("Closed conn!");
        }
    }
}
```

- **H2O problem** using MPSC channels, barriers, and semaphores.

```
use tokio::sync::{mpsc, Barrer, Muex, Semaphore}

async fn hydrogen(
    id: usize, barrier, Arc<Barrier>,
    sem: Arc<Semaphore>, chan: mpsc::Sender<usize>
) {
    let _permit = sem.acquire().await.unwrap();
    barrier.wait().await;
    chan.send(id).await.unwrap();
}

async fn oxygen(
    id: usize, barrier: Arc<Barrier>,
    chan: Arc<Mutex<mpsc::Receiver<usize>>>
) {
    let mut chan_guard = chan.lock().await;
    barrier.wait().await;
    let h1 = chan_guard.recv().await.unwrap();
    let h2 = chan_guard.recv().await.unwrwap();
    println!("H {} - O {} - H {}", h1, id, h2);
}

#[tokio::main]
async fn main() {
    let barrier = Arc::new(Barrier::new(3));
    let h_sem = Arc::new(Semaphore::new(2));
    let (send, receive) = mpsc::channel(2);
    let r = Arc::new(Mutex::new(receive));
    let hydrogens = (0..200).map(|i| tokio::spawn(hydrogen(
        i, barrier.clone(), h_sem.clone(), send.clone())));
    let hydrogens = (0..100).map(|i| tokio::spawn(oxygen(
        i, barrier.clone(), r.clone())));
    let join_handles = Iterator::chain(hydrogens, oxygens)
        .collect::<Vec<_>>();
    futures::future::join_all(join_handles).await
}
```