CS3211

AY22/23 Sem 2

github.com/SeekSaveServe

Lectures

L1: Introduction to Concurrency

Concurrency
- Concurency is pervasive when modern computers have several cores and types of memory
- $\geq$ 2 activities making progress at the same time (overlaping time periods)
- Involves interleaving of instructions from different activities

Parallelism
- $\geq$ 2 processes executing and making progress *simultaneously*
- Hardware dependent: requires hyperthreading (SMT), or multi-core and hardware threads

Processes Vs threads
- Independent Vs Shared memory (address) space
- Both use independet stack
- Expensive Vs Cheap context switch
- OS facilitated vs Non OS facilitated inter-process/thread Communication
- Expensive (copy on write mediates this somewhat) Vs Cheap creation

Interrupts
- Asynchronous (independent to program execution)
- Used by OS to interact with the programme
- Triggered by external events (e.g. I/O, timer, hardware failure)

Exceptions
- Synchronous (dependent on program execution)
- Used by process to interact with the OS
- Triggered by process error (e.g. underflow, overflow)

User thread
- Library created, linked to one kernel thread

Race condition
- Outcome depends on reltive ordering of operations on ge 2 Threads
- a flaw that occurs when the timing or ordering of events affects a program's correctness

Data Race
1. $\geq$ 2 concurrent threads concurrently access a shared resource without Synchronisation / fixed ordering
2. At least one modifies shared resource
3. Causes undefined behaviour

Mutex
- Creates critical section can be treated as a large atomic blocks
- Only one thread at a a time
- Supported by a hardware instruction (CAS, test and set etc)
- **Properties:** Mutex, progres, bounded wait, performance
- Provides **serialisation** (less concurrency)

Critical section
- Safety: nothing bad happens
- Liveness: Something good (progress) happens
- Performance: depends on aggregate performance of all threads

Locks
- Primitive that is provided b the hardware, minimal semantic
- E.g. Test and set

Deadlock iff
1. Mutex: One resource held in a non-shareable state
2. Hold and wait: One proces holding one resource and waiting for another resource
3. No-preemption: Resource and critical section cannot be aborted externally
4. Circular wait
5. Note: Lock free can deadlock

Dealing with deadlock
- Prevention: Eliminate one of the above conditions (E.g. hold all locks at the start)
- Detection and recovery: Look for cycles in dependencies (E.g. wait for graph)
- Avoidance: Control allocation of resources

Starvation
- One process cannot progress becausee another process is holding on a resource it needs
- Side effect of scheduling algorithm
- Wait-die and wound-wait are possible solutions, if priority of processes is preserved

Advantages of concurrency
- Performance
- Separation of concerns

Disadvanageous of concurrency
- Maintenance and debugging

Task parallelism
1. Do the **same type of** work faster
2. Task dependency graph can be parallel
3. Make tasks specialists: Same type of tasks are assigned to the same thread
4. Divide a sequence of tasks among threads to solve complexed task
5. **Pipeline:** 1 type of thread for one phase of execution

Data parallelism
1. Do **more work** in the same amount of time
2. Divide data to chunks and execute by different threads
3. Embarasingly parallel tasks

Challenges of concurrency
1. Finding enough parallelism: Amadahl's law
2. Granularity of tasks
3. Locality
4. Coordination and Synchronisation
5. debugging
6. Performance and monitroring

L2: Tasks, threads, synchronisation in modern C++

History of CPP
- 1998: No support for multithreading
  1. Effects of language model are assumed to be sequential and there are no established memory model
  2. Different libraries used different memory models
  3. Execution threads were not acknowledged
- 2011: C++11
  1. Standard threads are implemented
  2. Thread aware memory model.Do not rely on platform specific extensions to guarantee behaviour
  3. Atomic operations library, class to manage threads, protected shared data etc.

Four ways to manage threads

1. Declare a function that returns a thread

```
void hello() {
    std::cout << "Hello Concurrent
        World\n";
}
int main() {
    std::thread t(hello);
    t.join();// existing thread waits for
        t to finish
}
```

2. Thread with a function object

```
class background_task {
public:
    void operator()() const {
        do_something();
        do_something_else();
    }
};
/* Calleable object */
background_task f;
std::thread my_thread(f);
```

- $std :: my\_thread(background\_task())$ declares a function that takes a single parameter (type *f() $\rightarrow$ object)
- This is not the same as using a function object!

3. Threads with a lambda expression (local fn instead of a calleable object)

```
std::thread my_thread([]{
    do_something();
    do_something_else();
});
```

Wait
- Uses join() on the thread instance exactly once
- Use joinable to check
- Local variables do not go out of scope
- Blocking

Detach()
- Local variable passed might go out of scope and 'disappear' during runtime, causing invalid access for the detached thread
- Example

```
void oops(){
    int local_state = 0;
    /* Reference passed might become
        invalid */
    func my_func(local_state);
    std::thread my_thread(my_func);
    my_thread.detach();
} /*oops ends here and local_state will
    be destroyed */
```

- Not blocking

Passing arguments
1. by value $std :: threadt(f, 3, "hello")$
2. by reference $std :: threadt(f, 3, buffer)$
   - Buffer is a charbuffer that only gets converted to str when we call f
   - Hence it is possible for buffer to go out of scope
   - Fix: Use explicit cast $std :: thread(f, 3, std :: string(buffer))$
   - **Major issue** with passing by reference is that threads outside of the scope can use it in **unsafe** ways. E.g. Not using mutex on shared data, deletion etc
3. by copy

```
void update_data_for_widget(widget_id w,
    widget_data& data);
void oops_again(widget_id w) {
    widget_data data;
    /* a copy of data is passed */
    std::thread t(update_data_for_widget,
        w, data);
    display_status();
    t.join();
    /* changes made to the copy is not
        reflected to other threads */
    process_widget_data(data);
}
```

- Fix: use reference $std :: threadt(update_data for_widget, w, std :: ref(data))$

Ownership in C++
- Owner is an object containing a pointer to an object allocated by new for which the owner is responsible for deleting
- Every object on free store (heap, dynamic store) must have **exactly one** owner

C++ Resource Management
- For scoped objects, desctructor is implicit at scope exit
- Free store objects (created using new) requires explicit delete

RAII
- Binds the lifetime of a resource that must be acquired before use to the lifetime of an object

```
/* Handle interrupts using RAII */
void enqueue(Job job) {
    std::unique_lock lock{mut}; //
        constructor locks mutex
    jobs.push(job);// destructor unlocks
        mutex
}
```

Lifetime
- Lifetime begins when storage is obtained and its initialization is complete (except std::allocator::allocate)
- Lifetime ends when :
  - Non-class type (int): destroyed
  - Class type: When destructor is called
  - Reference: begins with initialisation and ends when destroyed. A dangling reference is possible.

Ownership of thread
- Moveable but not copyable

```
void some_function();
void some_other_function();
std::thread t1(some_function);
/* t1 no longer references the thread */
std::thread t2 = std::move(t1);
/* t1 now owns a new thread */
t1 = std::thread(some_other_function);
std::thread t3;
/* t3 owns the thread running some
    function */
t3 = std::move(t2);
/* t1 already owns a thread, this will
    trigger a runtime error */
t1 = std::move(t3);
```

- C++ compiler cannot catch this

- Ownerhship can be moved out of a function and moved into another function

```
/* Transferrring out of a function */
std::thread g() {
    void some_function();
    std::thread t(some_function);
    return t; // ownership transferred
        out of g()
}

/* Transferring into a function */
void f(std::thread t);
void g() {
    void some_other_function();
    std::thread t(some_other_function);
    f(std::move(t)); // ownership
        transferred into f()
}
```

Mutex in C++

- $std::lock_guard$ locks the mutex upon initialisation, unlocks upon destruction
- $std::lock_guard < std::mutex > some_mutex;$
- Group mutex and protected data together in a class rather than use global variables
- Never pass data or pointers (via returns, sotring in externally visible memory, as input to functions etc) when their usage is not guaranteed to be safe

Types of lock guards

- **lock guard** no manual lock, can lock many or one mutex at once without deadlock
- **Scoped lock** accepts and locks a list of mutexes. Can be unintentionally initialized without a mutex
- **unique lock** manual unlock, defers locking using $std::defer_lock$, only single mutex.

Condition Variable

- Use condition variables to wait for a an event to be triggered by another thread
- Avoids busy waiting

```
std::condition_variable.wait(lock,
    []{return predicate});
```

- if condition is satisfied, returns
- unlocks the mutex and places the thread in block state if condition is not satisfied
- $std::condition_variable.notify_one();$ to notify one thread waiting on the cond

Spurious Wake

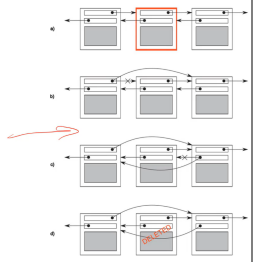
- Thread wakes up from waiting, but is blocked again as the resource required is not available
- Leads to unnecessary context switching
- Use conditionals to prevent spurious wake

Shared DS - Invariants

- Often broken during an update
- **Case study: Doubly LL**

*Invariant is broken during the delete

a) Identify the node to delete: N.
b) Update the link from the node prior to N to point to the node after N.
c) Update the link from the node after N to point to the node prior to N.
d) Delete node N.



- invariant is temporarily broken during an update, and we need to prevent objects from accessing the DS during this time

L3: Atomics and Memory Model in C++

Reordering of operations

- Compiler may reorder (potentially conflicting) actions for performance
- Not visible to programmers

As-if rule

- Reordering is allowed as long as:

1. At termination, data written to files is exactly as if the program was executed as written (same final state)
2. Prompting text that is sent to interactive devices will be shown before the program waits for input
3. Programs with undefined behaviour is exempted from these rules.

Multi-threading aware memory model

- Using synchronisation constructs (mutexes, barriers etc) should preclude the need for a memory model since they serialise threads
- Memory model gives us more flexibility and speed by getting us closer to the machine

Stucture of memory model

- Every object has a memory location, some occupy exactly one, some occupy many
- The changes in memory location / what is stored there affects other threads

Modification Order

- Compose of all writes to an object from all threads in the program
- MO varies between runs, each object has their own MO
- The programmer is responsible that threads agree on the MO (if not, race condition happens)

MO - Requirements

- The MO of each object is monotonic within a thread
- But the relative ordering of MO of different objects is not guaranteed

MO - Building Blocks

- **Sequenced-before (SB)**
  - Each lines of code in a thread is sequenced before the next line
  - There is **NO** sequenced before in a statement with many function calls
- **Synchronises-with (SW)**
  - Established by a $load$ from T_i reading T_j's $store$
  - Both T_i and T_j are synced with respect to the common value in the MO
- **Happens-before (HB)** When an operation happens before another operation due to SW or SB
- **Interthread Happens Before (IHB)**
  - When a $store$ in T_i established a sequenced before a $load$ in T_j, $sotre_i$ happens before $load_j$
  - IHB $\subseteq$ HB
- **Visible Side effects**
  - Side effect of write A on O is visible to a read B on O if:
    1. A HB B
    2. There is no other side effects to O that happens between A and B
  - If the side effect of A is visible to B then the longest contiguous subset of the side-effects to O (that B does not HB) is known as the visible sequence of side effects
  - Do not think of ordering, think in terms of side effects that are visible
- **Modification Order**

MO - Seq Const

- The default
- All threads must see the same ordering of operation
- Synchronises with a sequentially consistent load of the same variable that reads the value loaded
- Does not apply to atomic operations with relaxed ordering
- Performance penalty when working with weakly ordered machine instructions (common)
- Essentially a serailised monoversion - global total order enforced
- Only guaranteed for data-race free programs (which is difficult since C++ is not as safe as Rust)

MO - Relaxed

- Atomic operations don't conform with SW relationships
- Happens before still applies within the thread $\rightarrow$ monotonicity and SB within the thread is preserved
- No HB between load and store, different store operations from T1 can be viewed out of order by reads in T2
- T1: $x = 1, y = 0$. T2 can see y=0 without seeing x=1 since there is no SW between the two threads even though x=1 HB y=1 in T1.

MO - Acquire Release

- No total modification order, but there is a partial order
- Read - acquire updates about the memory order, load - release updates about the memory order
- A link between acquire and release acts like a barrier

MO - Mixing Models

- Seq const and Release Acquire: load and store of seq const behaves similar to release acquite
- any MO and relaxed: Relaxed behaves like relaxed but is bounded by the other more limiting MO

```
// T1
x.store(true, std::memory_order_relaxed);
y.store(true, std::memory_order_release);
// T2
while (!y.load(std::memory_order_acquire));
/* Never fires because acquire and release */
/* x.store HB y.store  & y.store SW y.load */
assert(x.load(std::memory_order_relaxed));
```

Atomic Operations

- Compiler ensures necesary synchronisation is in place and enforces MO
- Atomic ops are indivisible
- Atomic load loads either the initial value or the value stored by one of the modifications (cannot be half-done)
- Can be lock free or be implemented using mutex (which wipes off performance gains)
- Not necessarily race free

L4: Testing and debugging in C++

Concurrency related bugs

- Unwanted blocking: Deadlock, livelock, blocking while waiting on I/O
- Race conditions:

1. Data races: Undefined behaviour due to unsynchronised access to a shared memory locaiton. Obseravable.
2. Broken Invariants:
   - Dangling pointers: another thread deleted the data being accessed
   - Random memory corruption: Inconsitent values being read due to partial updates
   - Double free: Two threads pop the same value from a queue / deleting the same memory address twice - causes possible memory leak
   - Use After Free - Reading or writing after a memory has been freed
   - Uinitialised variables - Reading from a variable that has not been initialised
3. Lifetime issues:
   - Thread outlives data
   - Call to join skipped doe to an exception thrown

Techniques to locate concurrency bugs

- Look at the code
- Testing: Difficult to reproduce, Tests do not always fail (Heisenbug)

Guidelines for testing

- Run the smallest amount of code that could potentially demonstrate the bug to locate faulty code
- Do single threaded tests to verify the bug is concurrency related
- Run on sinlge core system to identify issues with interleavings

Test environment

- Number of threads: More threads increases the chance of deadlock (at least 2), contention (blocking while contending for shared resources, degrades performance), overhead
- Architecture
- Number of cores
- Having memory fences and barriers to sync threads

Designs for testability

- Responsiblity for each function and thread should be clear

- Check that library calls are thread safe (e.g. do they use internal states to ensure correctness?)

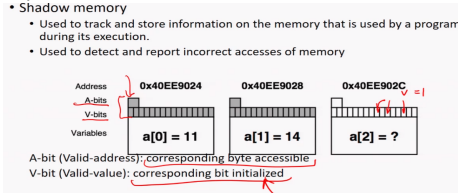
Techniques for testing

1. Stress testing
2. Use special implementation of synchronisation primitives (e.g. log when mutexes are locked, unlocked)
3. **Scalability** does the speedup scale when the number of threads increase? Contention to

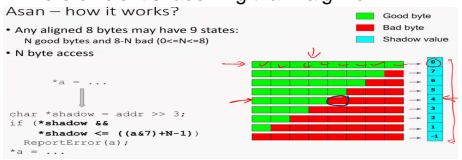
Debugging Tools

- Identify bugs:

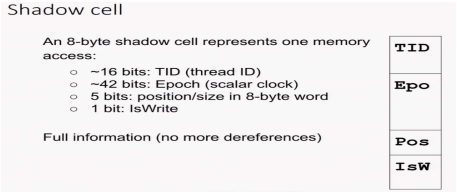
1. Valgrind (dynamic instrumentation)
   - Shadow memory: track and store information on the memory that is used by a program during its execution. A bits and V bits must match for valid.
   - 20x slowdown
   - Provides more details than sanitizers despite its performance overhead



2. Helgrind (dynamic instrumentation)
   - 100x slow down
   - intercepts function calls to functions and instruments
   - Detects: misuses of POSIX threads API, potential deadlocks (checking cyclic lock acquisition), data races (checks existence of HB between memory accesses)
3. Sanitizers (compilation-based approach)
   - 5-10x overhead (lmao just use rust)
   - $-fsanitize = address$ or $-fsanitize = thread$ or $-fsanitize = memory$ (to catch unitialised memory)
   - Output is verbose (unlike rust compiler)
4. Adress Sanitizers (compilation-based approach)
   - 2x slowdown, 3x overhead
   - More efficient shadowing than valgrind



5. Thread Sanitizer
   - 5-10x slow down, 5-15x overhead
   - Function entry/exit and memory access are logged
   - An 8-byte shadow cell represents one memory access
   - Epoch: time of access
   - Pos: location accessed
   - If there is an overlap in Pos, we check epoch for evidence of HB. If no HB, then there might be a data race
   - Use graph-based deadlock detection



L5: Concurrent DS in modern C++

Goal

- Multiple threads can access the same DS concurrently
- The scope of concurrency (all operations, some operations, one operation) depends
- Each thread has a self-consistent view of the DS
- **Broken invariants** should not be visible E.g. delete() of DLL
- Avoid race conditions
- Handle exceptions, prevent exceptions from exposing broken invariants. E.g. UAF when a thread aborts after freeing
- **Thread safe**
  - No data is lost or corrupted
  - All invariants are kept
  - No problematic race conditions

Problems with mutex

- Prevents true concurrent access to DS - Serialisation
- Possible for deadlocks (well, same for lock-free but harder)

Concurrency while calling functions

- Constructors and destructors require exclusive access to the DS - users should not access DS before construction is complete and after desctruction
- Swap(), assignment, copy(): Can they be used concurrently with other operations in the DS?

Design Principles

- Smaller the protected region, the better - fewer serialised region
- Provide opportunities for concurrency to threads accessing a thread safe DS - what can be called concurrently?
- One type of operation can be performing one type of operation exclusively from another type of operation (reader writer) - consider shared mutex
- Or allow different types of operations to happen concurrently but disallow the same type to be used by concurrent threads

Maximising concurrency

- Use different mutexes to protect different parts
- Give more concurrency to more frequent operations

Example: threadsafe stack

1. Exception Safety - safe!

```
std::shared_ptr<T> pop() {
    std::lock_guard<std::mutex> lock(m);
    /* This is safe */
    if (data.empty()) throw empty_stack();
    /* Potential out of memory error */
    /* This is fine since the mutex will
       be unlocked during exception */
    std::shared_ptr<T> const res(
std::make_shared<T>(std::move(data.top())));
    value = std::move(data.top());
    data.pop();
    return res;
}
```

2. Work is serialized for the DS - low concurrency
3. Should use a monitor to allow waiting for an item to be added

Example: threadsafe Queue

1. Exception Safety: Suppose a thread is woken up by the monitor and that an exception occuring after this point will cause nothing to be popped from the queue.
2. And other threads waiting on the condition variable is not able to be notified of the non-empty queue

```
void wait_and_pop(T & value) {
    std::unique_lock<std::mutex> lock(m);
    data_cond.wait(lock, [this]{return
        !data.empty();});
    std::shared_ptr<T> res(
/* Exception here is problematic */
std::make_shared<T>(std::move(data.top())));
    value = std::move(data.top());
    /* Important to pop after moving */
    data.pop();
    return res;
}
```

3. Fix 1 Notify All: Works ... but cause spurious wakes
4. Fix 2: Put the shared pointer in the queue directly, so we can obtain a shared pointer directly by popping from the queue
5. Shared pointer helps us to handle deallocation

```
class threadsafe_queue {
    private:
    std::mutex m;
    std::queue<std::shared_ptr<T>>
        data_queue;
    std::condition_variable data_cond;
    public:
    ...
void push(T new_value) {
    /* Creation of new data takes place
       outside mutex */
    /* This is exception safe */
    std::shared_ptr<T> data(
std::make_shared<T>(std::move(new_value)));
    std::lock_guard<std::mutex> lock(m);
    data_queue.push(data);
    data_cond.notify_one();
}
}
```

6. Share-pointers are exception safe
7. The creation now takes place outside the mutex - improves performance
8. However, using the standard container and mutex limits concurrency as the queue is either protected or not
9. For a more fine-grained locking, we need to write a customised DS

Example: threadsafe stack with fine-grained locks

- **Node**

1. Pointers to nodes should only be modified by one thread, so we use an unique poitner

```
struct node {
    std::shared_ptr<T> data;
    /* Only one copy of next */
    std::unique_ptr<node> next;
    node(T data_):
            data(std::move(data_))
    {}
};
```

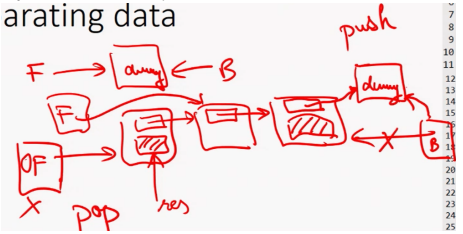
- **Push - attempt 1**

1. Modify the of the queue if it is empty
2. Else we modify the back
3. Using a lock for checking front and back is problematic as we need to lock both mutexes if the queue is initially empty (front=back, potential for deadlock if two threads tries to push concurrently). This also serialises the queue

```
void push(T new_value) {
    std::unique_ptr<node> p(new
        node(std::move(new_value)));
    node* const new_back = p.get();
    std::lock_guard<std::mutex> lk(mut);
    if (back) {
        back->next = std::move(p);
    } else {
        front = std::move(p);
    }
    back = new_back;
}
```

- **Push - attempt 2**

1. Ensure that there's always $\geq$1 node in the queue to separate the node being accessed at the front from the node being accessed at the back
2. Empty queue: front and back point at a dummy node
3. No race on front.next and back.next (but with additional layer of indirection)



- With the use of dummy nodes, we can do more fine grained locking since we can now lock the front and back separately without the chance of Deadlock
- Pop and push can now occur concurrently

```cpp
template<typename T>
class threadsafe_queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };
    std::mutex front_mutex;
    std::unique_ptr<node> front;
    std::mutex back_mutex;
    node* back;
    node* get_back()
    {
        std::lock_guard<std::mutex> back_lock(back_mutex);
        return back;
    }
    std::unique_ptr<node> pop_front()
    {
        std::lock_guard<std::mutex> front_lock(front_mutex);

        if(front.get()==get_back())
        {
            return nullptr;
        }
        std::unique_ptr<node> old_front=std::move(front);
        front=std::move(old_front->next);
        return old_front;
    }
public:
    threadsafe_queue():
        front(new node),back(front.get())
    {}
    threadsafe_queue(const threadsafe_queue& other)=delete;
    threadsafe_queue& operator=(const threadsafe_queue& other)=delete;
    std::shared_ptr<T> try_pop()
    {
        std::unique_ptr<node> old_front=pop_front();
        return old_front?old_front->data:std::shared_ptr<T>();
    }
    void push(T new_value)
    {
        std::shared_ptr<T> new_data(
            std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_back=p.get();
        std::lock_guard<std::mutex> back_lock(back_mutex);
        back->data=new_data;
        back->next=std::move(p);
        back=new_back;
    }
};
```

- This can be even more fine grained by having a per-node mutex, which establishes a SW between push and pop threads
- However this doesn't really make sense (unless we are supporting search etc) as pop and push only accesses the front and back of the queue
- We also have to use raw pointers since the mutex is in the Node and we cannot delete the Node before unlocking (otherwise UAF)

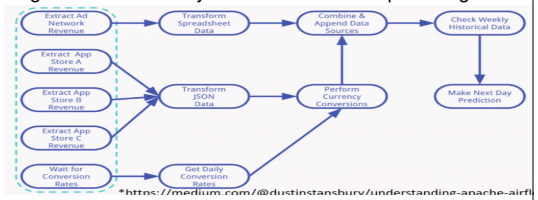
Lock-free concurrent DS
- Non-blocking Mutexes: Spinning lock with test and set
- A blocked thread is suspended by the OS
- Non-blocking DS:

1. **Obstruction-free:** if all threads paused, then any thread will complete in a bounded number of threads
2. **Lock-free:** if multiple threads are operating on a DS, then after a bounded number of steps, one of them will complete (note: deadlock still possible)
   - **Pros**
   - Enable max concurrency; some thread makes progress with every step
   - Robust: One thread dying doesn't affect the other threads using the DS, threads will not be excluded from accessing the DS
   - Need to ensure that the invariants are upheld, and that changes become visible to other threads in the correct order
   - **Cons**
   - live locks are possible (changes made by one thread requires the others to restart)
   - Decrease overall performance even though individual waiting time is reduced
   - Atomic operations can be expensive
   - Hardware need to use more expensive operations to synchronise data
3. **Wait-free:** every thread will complete in a bounded number of steps, even if other threads are operating on the DS (note: starvation still possible)
   - Lock free with loops (CAS) can result in one being subjected to starvation
   - E.g. A spins, but B progresses and sets the condition to false again
4. wait-free $subseteq$ lock-free $subseteq$ obstruction-free
5. lock-free is not always wait-free

Contention and Cache Ping Pong
- Change made by one thread takes time to propagate to the other cores
- This contention makes other threads wait for the change to propagate across memory
- Accessing data from the cache line with multiple threads can cause false sharing (hard to catch)
-

L6: Concurrency in GO

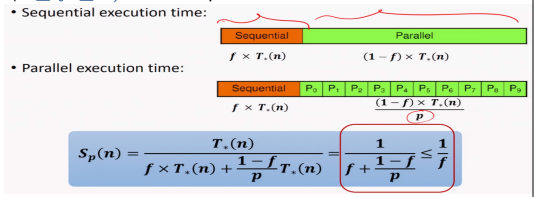
Task Concurency Graph
- Node: Task, value is the expected execution time
- Edge: Represent control dependency between tasks (v wait-for u)
- Critical path length: slowest completion time
- Degree of concurrency: Total work / critical path length



Speed up gained from concurrency

$S_p(n) = \frac{T_{bestseq}(n)}{T_{parallel}(n)}$, we aim to gain a S_p of p, where p is the number of processors

Amadahl's Law
- Speed up of parallel execution is limited by the fraction of the algorithm that cannot be parallelised
- f$(0 \leq f \leq 1)$ is the sequential fraction



About GO
- Statically typed
- Compiled PL
- More memory safety, garbage collection, CSP style concurrency than C

Go routeins
- Runs on OS threads - any type of threads the OS provides (go routines are tasks, not threads)
- A function running independently
- Cheaper than threads (a few kbs, 3 instructions to create) + lightweight
- Follow the fork-join model
- Not garbage collected - programmers to prevent leaks
- Goroutines are multiplexed to OS threads: allows scheduling and goroutines to be multiplexed to threads.
- Decouples concurrency from parallelism
- Go routine blocks $\rightarrow$ thread blocks. But other goroutines can still run

Communicating Sequential Processes (CSP)
- Concurrency: structure a program by breaking it into pieces that can be executed indepedently
- Go Routines:
  - A function running independently
- Communication: Coordinate the independent executions
- Channels:
  - Read: ¡- Channel
  - Write: channel ¡-
  -

Wait groups
- A blocking call that waits for n goroutines to finish (n can be incremented)

```go
var wg sync.WaitGroup
for _, salutations := range
    []string{"hello", "greetings", "good␣
    day"} {
    wg.Add(1)
    go func(salutation string) {
        # called when scope ends
        defer wg.Done() # decrements
            counter of wg
        # Salutation is passed by value
        fmt.Println(salutation)
    }(salutations)
}
wg.Wait()
```

Shared memory
- Similar to C++, need to synchronise access to shared memory
- Use channels instead of modifying shared memory

Channels
- Channel is a reference to a place in memory where the channel resides
- Typed
- Bi directional or unidirectional
- Blocking
  - Writing to full Channels
  - Reading from empty channel
  - can cause deadlock
  - Reading from closed channel returns default value; nonblocking
- Declaration:
  - Bi directional

```go
// Unbuffered Channels
var c chan int
// Buffered Channels
datastream := make(chan int, 5)
```

  - unidirectional

```go
var writer chan<- int // write only
var reader <-chan int // read only
datastream := make(chan int)
writer = datastream
reader = datastream
```

Synchronising using channels
- Ranging over channel, channel closes when loop ends

```go
for i := range intStream {
    // do something
}
```

- Close unbuffered channel

```go
begin := make(chan interface{})
var wg sync.WaitGroup
for i := 0; i < 5; i++ {
    wg.Add(1)
    go func(i int) {
        defer wg.Done()
        <-begin
        fmt.Printf("%v␣has␣begun\n", i)
    }(i)
}
fmt.println("Unblocking␣goroutines...")
// This allows default values to be
    printed
close(begin)
wg.Wait()
```

Channel behaviour

Operation	Channel state	Result
Read	nil	Block
	Open and Not Empty	Value
	Open and Empty	Block
	Closed	<default value>, false
	Write Only	Compilation Error
Write	nil	Block
	Open and Full	Block
	Open and Not Full	Write Value
	Closed	**panic**
	Receive Only	Compilation Error
close	nil	**panic**
	Open and Not Empty	Closes Channel; reads succeed until channel is drained, then reads produce default value
	Open and Empty	Closes Channel; reads produces default value
	Closed	**panic**
	Receive Only	Compilation Error

Buffered channels
- Delays the problem of a blocking (full/empty) channel

Guidelines: Ownership of a channel (NOT checked by compiler)
- Owner is the goroutine that instantiates, writes and closes
- (unidirectional) Owner writes, utilisers read

Owner should	Consumer should
• Instantiate the channel	• Know when a channel is closed
• Perform writes, or pass ownership to another goroutine	• Responsibly handle blocking for any reason
• Close the channel	
• Encapsulate 1.-3. and expose them via a reader channel	

- **Increases safety**

1. The owner will not deadlock by writing to a nill channel
2. The owner will not panic by closing a closed channel
3. The owner will not panic by writing to a closed channel since the owner closes the channel
4. The owner will not close the channel more than once because it closes the channel
5. Since only owner writes, the type checker can prevent improper writes at compile time

Select

- Blocks until a branch becomes available, all branches are considered simultaneously
- Use cases:

1. Multiple channels have something to read
   - Peudo-randomly selects a channel to read from. Supposedly uniform

```
c1 := make(chan interface{});
    close(c1) // nonblocking
c2 := make(chan interface{});
    close(c2) // nonblocking
var c1Count, c2Count int
select {
    case <-ch1:
        c1Count ++
    case <-ch2:
        c2Count ++
}
```

2. Channels are not ready
   - Timeout channels that are never ready

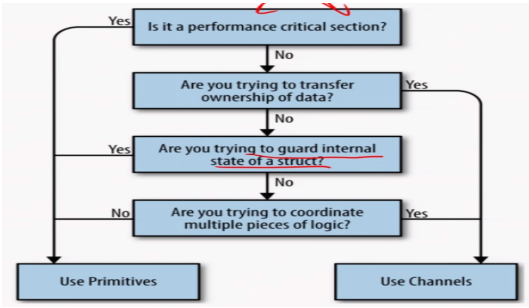
```
var c1 <- chan int
select {
    case <- c:
    // times out after 1 second
    case <- time.After(1 *
        time.Second):
        fmt.Println("Timed out")
}
```

- For-select loop
  - Allows a Go routine to make progress while waiting for other go routines

```
done := make(chan interface{})
go func () {
    for {
        select {
            // work is done!
            case <- done:
                return
            default:
        }
        // do non-preemptable work
    }
}() // this () calls the function
    immediately after defining
```

Sync package

- Do not communicate by sharing memory. Instead, share memory by communicating.



Memory Model - GO

- Similar to C++, except theres no atomics
- Synchronised before:

1. Send is synced before a corresponding receive from another thread
2. Closing of a channel is synced before reads from the closed channel (value 0)
3. Receive from unbuffered channel is synced before the send on that channel completes
4. The kth receive on a channel with capacity C is synced before the (k+C)th send on the same channel
5. *Go* statement that starts a gorountine is synced before the go routine's execution
6. The exit of a gopher is **NOT** guaranteed to be synced before any code after the go statement

L7 Concurrency Patterns in GO

Patterns - Confinement

- Achieve safe operation through synchronisation (mutex / channels)
- Safe concurrency with good performance - immutable data, data protected by confinement
- **Ad hoc confinement**
  - Data is modified only from one gopher, even though the data is accessible by multiple
  - Needs static analysis to ensure safety (Exclusive ownership)
  - Easily broken
- **Lexical confinement**
  - Restric access to shared location
  - Achieved by exposing only the reading or writing handle to the producer/consumer
  - Expose only a slice of the DS (e.g. array)

Pattern - For-select loop

- Context - sending iteration variables out on a channel
- Looping and waiting to be stopped (by context or done channel)

Goroutine leakage

- Goroutines are detached and not garbage collected
- Creator of goroutines should also destroy them
- Passing 'nil' as a channel will cause the gopher to run indefinitely
- Solutions:
  - Pass a done channel, close it in the creator to signal the gopher to stop

Pattern - Error handling

- State goroutine maintains complete state of the program
- All goroutines send their error to a state goroutine that handles this
- Couple the potential error with potential result
- E.g. Http handling

```
type Result struct {
    Error error
    Response *http.Response
}

for _, url := range urls {
    result := Result{Error: nil,
        Response: nil}
    go func() {
        resp, err := http.Get(url)
        result = Result{Error: err,
            Response: resp}
        response <- result
    }()
    select {
        case <-done:
            return
        /* Owner of results will handle
            this */
        case results <- result:
        default:
    }
}
```

Pattern - Pipeline

- A series of stages connected by channels, each stage performs the stage function
- Inbound and outbound channels passes data from one channel to the next
- The first stage only has an outbound channel
- The last stage only has an inbound channel
- Intermediate stages have multiple inbound and outbound channels
- Aims to achieve separation of concerns

Pipleine - Efficiency

- Work and resources should be divided among stages s.t they all take the same time to complete their tasks
- Fan-out to decrease the processing time for a stage
- Use I/O and multiple CPUs to process different streams of data
- Not obviously faster than task pool unless optimised
- Pipeline is better if there is a cap on a specific resource that is needed by all task pool (at different times)
- E.g

```
done := make(chan interface{})
defer close(done)
intStream := generator(done, 1, 2, 3, 4)
pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)
for v := range pipeline {
    fmt.Println(v)
}
```

  - generator, multiply etc are functions written using the for-select paradigm

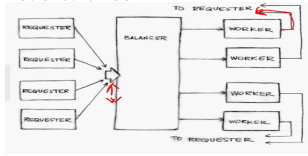
Pattern - Fan-out, Fan-in

- Problem: Stages in pipeline might be slower than the other and they might benefit from parallelism
- Fan-out: Start multiple gophers to handle inputs
  - The stage should not rely on values that has been calculated before in preceding stages
  - The order of the concurent copies does not matter (is not maintained)

```
numFinders = runtime.NumCPU() // Max
    level of concurrency
// Channel of writer channels
finders := make([] <- chan int,
    numFinders)
for i := 0; i < numFinders; i++ {
    go func() {
        for i := range numFinders {
            // Fan-out
            finders[i] =
                primeFinder(done,
                inputStream)
        }
    }
}
```

- Fan-in: Combine multiple results into one output channel
  - Multiplexing/joining multiple streams of data into a single data
  - Order will not be maintained

Load balancer



- Balance load among all workers based on the amount of request in the queue of each worker
- The sending back of results is not done through the load balancer; a separate channel
- Requst

```
type Request struct {
    fn funct() int // The operation to
        perform
    result chan int // The channel to
        return the result
}
```

- Requester

```
func requester(work chan<- Request) {
    c := make(chan int) // read result
        from c
    for { // while loop
    /* For the balancer to proess */
        case work <- Request{workFn, c}:
        result := <-c // wait for worker
        furtherProcess(result)
    }
}
```

- Worker
  -

```
/* Receiver of the method, basically this
    is a fn of the worker struct */
func (w *worker) work(done chan *Worker) {
    for {
        req := <-w.requests
```

```
            req.c <- req.fn() // call fn and
                send result
            done <- w // we've finished this
                request
        }
    }
```

- Balancer - DS

```
type Pool []*worker // track the workers

type Balancer struct {
    pool Pool
    done chan *worker // shared among all
        workers
}
```

- Dispatch - a min heap of channels

```
func (p Pool) Less(i, j int) bool {
    return len(p[i].requests) <
        len(p[j].requests)
}

// Send request to worker
func (b *Balancer) dispatch(req Request) {
    // Grab the least loaded worker
    w := heap.Pop(&b.pool).(*worker)
    // Send request to worker
    w.requests <- req
    // Put worker back into the pool
    heap.Push(&b.pool, w)
}
```

- Completes

```
func (b *Balancer) completed(w *worker) {
    w.pending --
    // Remove worker from pool
    heap.Remove(&b.pool, w.index)
    // Put it into heap
    heap.Push(&b.pool, w)
}
```

- Balance (can be pipelined, see T6 3.2)

```
func (b *Balancer) balance(work chan
    Request) {
    for {
        select {
            case req := <-work:
                b.dispatch(req)
            case w := <-b.done:
                b.completed(w)
        }
    }
}
```

Concurrent Balancer

- Use pipelining to make the balancer more concurrent
- Some requests can be starting at the dispatch stage
- Othe requests can be waiting at the completed stage
- Use a for-select pattern for this

Work Stealing

- Steal work from the queue of the slowest (longest queue) worker
- We steal from the back of the queue (in case there is dependence on the requests)
-

GO Runtime - work stealing

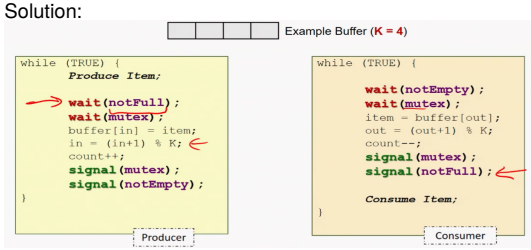
- (Fork Point) Add tasks to the tail of a deque of a thread
- If the thread is idle, steal the last job added to the deque associated with some other random thread
- (Join Point) At a join point that cannot be realised yet (waiting for some forked thread to complete), pop work off the tail of the thread's own deque (and process that)
- If the deque is empty, steal work from head of a random thread's deque
- Work stealing does stall the join, sometimes GO does continuation stealing instead

L8: Classical Synchronisation Problem in C++ and GO

Problem	CS problem
Barrier	Wait until threads/processes reach a specific point in the execution
Producer-consumer	Model interactions between a processor and devices that interact through FIFO channels.
Readers-writers	Model access to shared memory
Dining philosophers	Allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
Barbershop	Coordinating the execution of a processor.
FIFO Semaphore	Needed to avoid starvation and increase fairness in the system.
H2O	Allocation of specific resource to a process.
Cigarette smokers	The agent represents an operating system that allocates resources, and the smokers represent applications that need resources.

Producer-Consumer Problem

- Specifications:
  - Producers put in a shared bounded buffer if not full, consumers read from it if not empty
- Solution:



- Concurrent read, exclusive read. Categorical starvation of writer is possible

```
Writers                          Readers
roomEmpty.wait ()                mutex.wait ()
    #critical section for writers readers += 1
roomEmpty.signal ()              if readers == 1:
                                     roomEmpty.wait () # first in locks
                                 mutex.signal ()
                                 # critical section for readers

                                 readers -= 1
                                 if readers == 0:
                                     roomEmpty.signal () # last out unlocks
```

- Light switch: Abstracts out the shared lock for the reader

```
counter = 0
mutex = Semaphore (1)
lock (semaphore):
    mutex.wait ()
    counter += 1
    if counter == 1:
        semaphore.wait ()
    mutex.signal ()
unlock (semaphore):
    mutex.wait ()
    counter -= 1
    if counter == 0:
        semaphore.signal ()
    mutex.signal ()
```

- Starvation free solution:

```
Writers                          Readers
turnstile.wait ()                turnstile.wait ()
roomEmpty.wait ()                turnstile.signal ()
    # critical section for writers readSwitch.lock ( roomEmpty )
turnstile.signal ()                  # critical section for readers
roomEmpty.signal ()              readSwitch.unlock ( roomEmpty )
```

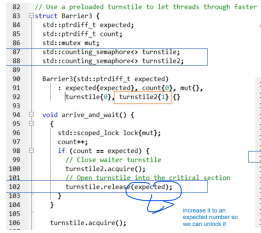
- Prioritise Writer:

```
readSwitch = lightswitch();
writeSwitch = lightswitch();
noReaders = semaphore(1);
noWriters = semaphore(1);

reader() {
/* Waiting for writers to be done */
wait(noReaders);
/* Writers cannot enter */
    readswitch.lock(noWriters);
signal(noReaders);
    # critical section
readSwitch.unlock(noWriters);
}

writer() {
/* Immediately acquires no readers so
    writers have priority */
writeSwitch.lock(noReaders);
    wait(noWriters);
    # critical section
    signal(noWriters);
writeSwitch.unlock(noReaders);
}
```

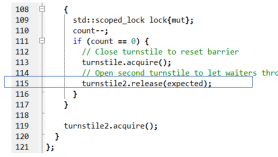
- This is implemented in C++ as a **shared_lock** and **unique_lock**
- GO has something similar: readLock and writeLock

Barrier

- All threads must stop at a common point before proceeding, can be reusable (barrier) or single use (latch)
- std::barrier, std::latch in C++
- E.g. std::barrier arrivalPoint(size) ... arrivalPoint.arrive_and_wait()
- sync.WaitGroup in GO is a latch, we can use 2 of them to make a barrier
- C++ implementation
  - The naive version fails because context switch can happen right before counter == N, which causes multiple threads (that were context switched out after counter ++) to signal the switch (another way to fail is to have 1 thread lap everyone else between the first barrier unlocks and second barrier unlocks)
  - The solution is to add a second turnstile (initialised as 1) to guard the turnstile1.signal, such that only one thread can signal it
  - But using mutex to increment turnstile1 one-by-one is slow
  - So we use a counting semaphore instead so we can raise the barrier by 1 thread!



- Lines 102 and 115: counting semaphor can be increased by expected to allow threads to pass



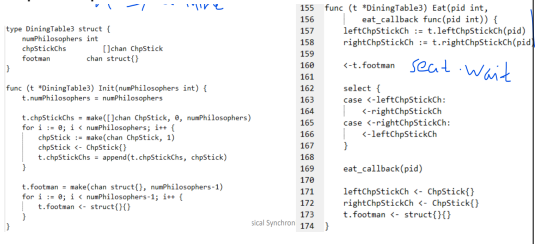
Dining Philosophers

- Specifications: N philosophers, N chopsticks
- Deadlock: All pick up left simultaneously
- Livelock: Put down left if right cannot be acquired
- Slap a mutex: Becomes sequential
- Scoped Lock(left, right): Acquire multiple mutexes in a deadlock free manner (deadlock avoidance), but as we have seen in CS3223, deadlock avoidance can lead to livelock
- GO's Mutex Free Solution: Use odd-even ring communication, odd numbered philosophers pick up left first, even numbered philosophers pick up right first
- This is the same as the right hander argument
- Tanenbaum's solution:

```
#define N 5
#define LEFT ((i+N-1)% N)
#define RIGHT ((i+1) % N)
#define THINKING 0
#define HUNGRY 1
#define EATING 2
int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher( int i ){
    while (TRUE){
        Think( );
        takeChpStcks( i );
        Eat( );
        putChpStcks( i );
    }
}
void safeToEat(i)
{
    if( (state[i] == HUNGRY) &&
    (state[LEFT] != EATING) &&
    (state[RIGHT] != EATING) ) {
    state[i] = EATING;
    signal(s[i]);
    }
}

void takeChpSticks( i )
{
    wait(mutex);
    state[i] = HUNGRY;
    safeToEat(i);
    signal(mutex);
    wait(s[i]);
}
```

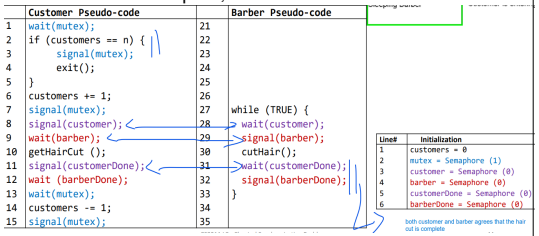
```
void putChpSticks(i)
{
    wait(mutex);
    state[i] = THINKING;
    safeToEat(LEFT);
    safeToEat(RIGHT);
    signal(mutex);
}
```

- Limited seats: Use a semaphore(N-1) to limit the number of philosophers that can eat at the same time



Barber Shop

- 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 waiting room is available, customer seats on one of the chairs
- If barber is sleeping, customer wakes him up
- If all chairs are occupied, customer leaves

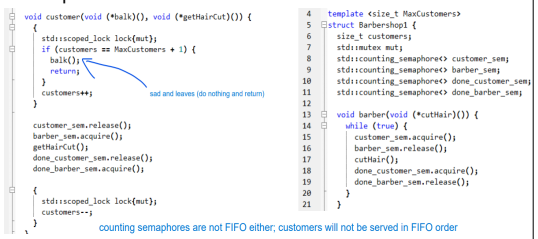


	Customer Pseudo-code		Barber Pseudo-code
1	wait(mutex);	21	
2	if (customers == n) {	22	
3	signal(mutex);	23	
4	exit();	24	
5	}	25	
6	customers += 1;	26	
7	signal(mutex);	27	while (TRUE) {
8	signal(customer);	28	wait(customer);
9	wait(barber);	29	signal(barber);
10	getHairCut ();	30	cutHair();
11	signal(customerDone);	31	wait(customerDone);
12	wait (barberDone);	32	signal(barberDone);
13	wait(mutex);	33	}
14	customers -= 1;	34	
15	signal(mutex);	35	

Line#	Initialization
1	customers = 0
2	mutex = Semaphore (1)
3	customer = Semaphore (0)
4	barber = Semaphore (0)
5	customerDone = Semaphore (0)
6	barberDone = Semaphore (0)

both customer and barber agrees that the hair cut is complete

- line 31-32 are important since we need customer and barber need to agree that the haircut is done
- GO implementation:



```
void customer(void (*balk)(), void (*getHairCut)()) {
  {
    std::scoped_lock lock(mut);
    if (customers == MaxCustomers + 1) {
      balk();
      return;
    }
    customers++;
  }

  customer_sem.release();
  barber_sem.acquire();
  getHairCut();
  done_customer_sem.release();
  done_barber_sem.acquire();

  {
    std::scoped_lock lock(mut);
    customers--;
  }
```

sad and leaves (do nothing and return)

```
template <size_t MaxCustomers>
struct Barbershop1 {
  size_t customers;
  std::mutex mut;
  std::counting_semaphore<> customer_sem;
  std::counting_semaphore<> barber_sem;
  std::counting_semaphore<> done_customer_sem;
  std::counting_semaphore<> done_barber_sem;

  void barber(void (*cutHair)()) {
    while (true) {
      customer_sem.acquire();
      barber_sem.release();
      cutHair();
      done_customer_sem.acquire();
      done_barber_sem.release();
    }
  }
```

counting semaphores are not FIFO either; customers will not be served in FIFO order

L9: Safety in Concurrent programming with Rust

Challenges in C++

- Unsafe memory usage (UAF, double free, danling pointers etc) - Go addresses this by using communication and not sharing memory
- Race condition (no data race in rust!), deadlocks etc

Fearless Concurrency in Rust

- Strong safety guarantees (no seg faults, no data races, expressive type system)
- Without compromising on performance (no garbage collector, no runtime)
- No runtime (In this context, "runtime" refers to a language runtime system, which is a piece of software that helps manage a program as it runs. This can include things like garbage collection, dynamic typing, exception handling, and other features that are typically associated with higher-level languages.)
- Memory safety like in garbage collected langauges (but without garbage collection)

Memory Safety in Rust

- Borrow checker statically prevents **Aliasing + Mutation** (...unsafe exists). Aliasing cannot coexist with mutation!
- Ownership prevents **Double Free**
- Borrowing prevents **UAF**
- No segfaults! Since compiler prevent unsafe memory usage.
- Compiler has really good error messages - unlike Tsan, Asan, and C++ compiler
- **No data race**: Sharing and mutation cannot happen at the same time! (if no unsafe used)
- **Race condition**: Still possible since no ordering + sharing || mutation is still possible

Type	Ownership	Alias?	Mutate?
T	Owned		Yes
&T	Shared reference	Yes	
&mut T	Mutable reference		Yes

Ownership - read trivia for a better take

- Prevents simultaneous mutation and aliasing
- The object that creates the object is the owner
- Declared with a mut keyword e.g. let mut book = Vec::new...
- Ownership can be transferred to other functios
- Does not use copy-constructor like C++
- clone() is a deep copy, while the copy-constructor in C++ is a shallow copy

Shared Borrow

- We can share a reference to an object without transferring ownership
- But the object cannot be mutated
- Shared borrowing one element in a vector will make the entire vector non-mutable (cannot push, pop etc)

Mutable Borrow

- Creates a mutable reference to an object
- But cannot be shared. We can pass the mutable reference to other functions only after the current one goes out of scope
- Prevents aliasing (two ptrs pointing to the same memory address) as we cannot access. This means that in a new scope, we cannot reference a variable that is already mutable in another scope.
- E.g.

```
let mut book = vec::new();
book.push(...); // success! Book is
    mutable
{
    let r = &mut book; // mutable borrow
        of book
    r.push(...); // success, reference
        can be mutated
    book.len(); // error, book is already
        borrowed!
}
book.push(...) // success, book is no
    longer borrowed
```

Library based concurrency

- Not built into the language (not even message passing)
- Library based - in std or other libraries

Thread creation

- thread::spawn(||) - spawns a new thread and returns a **JoinHandle**
- If the join handle is dropped, the thread is detached
- .join().unwrap() - join waits for the thread to finish and unwrap returns a result
- unwrap returns the panic if there is a panic

Transfer ownership through closure

- *move* converts any variable captured by reference or mutable reference to variables captured by value
- In Rust, the move keyword is used with closures to force them to take ownership of the variables **they use** from their environment
- Access to moved variables can trigger a **Use After Move** error
- Without move, closure would capture the reference to the variable (mutable borrow or shared borrow, compiler will infer), and they can still be useed after defining the closure
- **Use After Free Compile error**:

```
let mut dst = vec::new();
thread::spawn(|| {
    dst.push(3); // mutable borrow
})
dst.push(3)
```

- The compiler will regard the push in spawned thread as a potential UAF as the closure might be executed after the main thread has finished executing - dst will be dropped
- In this case we cannot use move either as it will trigger a UAM

Traits

- Reference Counter cannot be sent across threads because it has no **send** trait
- **Send** - transferred across thread boundaries
- **Sync** - safe to share references between threads (Type T is sync if and only if &T is send)
- **Copy** - Safe to memcpy
- **Clone** - create a deep copy of data
- For move, we need send and sync

ARC

- Allows only shared references
- References cannot be mutated
- Arc is destroyed when the reference to it goes to 0
- When an Arc containing an object is moved, the ownership of the object is with Arc, not the thread. Ref count ++
- A new Arc is cloned upon move.

Drop

- When we free, we relinquish ownership to an object
- drop(shared reference) only relinquishes the reference, not the object

MPSC, FIFO queue

- We can clone the writing head of MPSC multiple times but we cannot clone the reading end

Crossbeam

- **Scope Thread**: Gives a scope to a thread to give specific things to the thread rather than the whole environment
- Once the scope is ended, there is a guarantee that the ownership/lifetime of objects in the scope ends
- So the we can keep using the objects in the original scope in parts sequenced after the end of the scope

Exponential Backoff

- It is unhelpful to have a continuous poll / loop
- So we "backoff" for exponential amounts
- backoff.spin()
- backoff.snooze(), similar to yield

Rayon

- Data parallelism library
- Uses work stealing to balance load
- Need to commutative operations
- var.par_iter allocates 1 element to a thread in a round robin fashion

```
fn main() {
    let vec = init_vector();
    let max = AtomicI64::new(MIN);
    vec.par_iter().for_each(|n| {
        loop {
            let old = max.load(Ordering::SeqCst);
            if *n <= old {
                break;
            }
            let returned = max.compare_and_swap(old, *n, Ordering::SeqCst);
            if returned == old {
                println!("Swapped {} for {}.", n, old);
                break;
            }
        }
    });
    println!("Max value in the array is {}", max.load(Ordering::SeqCst));
    if max.load(Ordering::SeqCst) == MAX {
        println!("This is the max value for an i64.")
    }
}
```

Lifetime Rules

1. Each parameter that is a reference gets its own lifetime.
2. If there is exactly one input lifetime, that lifetime is assigned to all output lifetimes.
3. If there are multiple input lifetimes, but one of them is &self or &mut self, the lifetime of self is assigned to all output lifetimes.
4. If your code doesn't fit into these rules, you'll need to manually annotate lifetimes.

Tutorials

T1: Threads and Synchronisation

Why mutexes work - standard argument

- Define a critical section that contains all acesses to the shared resource
- Argue that mutex guarantees mutual exclusivity of threads - removes interleaving, data race precluded

Why mutexes work - theoretical argument

- Lock and unlock appears in a single total order
- Only one thread owns the lock at any pointer
- Unlock happens after lock, creating a synchronises with relationship between processes and **serialises the interleaving** - no concurrent access

Monitor

- Allows us to block until a condition becomes true
- The monitor has:

1. A mutex on the critical section
2. A condition variable
3. A condition to wait for

```
std::condition_variable cond;

Job dequeue() {
    std::unique_lock lock{mut};
    /* wait until there is a job */
    cond.wait(lock, [this]() { return
        !jobs.empty(); });
    Job job = jobs.front();
    jobs.pop();
    return job;
    }

void enqueue(Job job) {
    {
        std::unique_lock lock{mut};
        jobs.push(job);
    }
    /* notify one thread waiting on the
        condition variable */
    cond.notify_one();
    }
```

T2: Atomics in C++

Data races and undefined behaviour

- For undefined behaviour, the C++ compiler **will allow it to be compiled**
- Compilers are free to do any reordering and the answer cannot be predicted.
- E.g. GCC and Clang will return different runtime outputs as they compile it differently
- See snipet 1

Mutexes Vs Atomics

- Consider the following implementation of a counter

```
int counter = 0
std::mutex m;
void t1 {
    for (int i = 0; i < 1000000; i++) {
        std::lock_guard lock{mut};
        counter++;
    }
}

std::atomic<int> atomic_counter = 0
void t2 {
    for (int i = 0; i < 1000000; i++) {
        atomic_counter.fetch_add(1,
            std::memory_order_seq_cst);
    }
}
```

- The atomic version is much faser than the mutex version
- Mutex calls lock and unlock multiple times around the add operation, which compiles to more instructions and may cause the threads to sleep / fight for access
- The atomic version is a single instruction $lock\ addl\ \$1$, which is much faster

Memory order and performance

- Seq cst $\geq$ Acquire-Release $\geq$ Relaxed
- Seq cst's store uses $xchg$ (requires the processor to have exclusive access to shared mem) while the other orders allows x86 to use a simple mov instruction

Forcing ordering with $std::atomic<int>$

- Compiler reorders instructions to preserve visible side effects and optimise
- We can use atomics to force the compiler to preserve ordering
- But this also forces the compiler to use the more expensive $xchg$ instruction, since atomics use seq cst by default

Memory order - atomics

- Note: data races **is not possible** when atomics are used. If load and store are not atomic, then the output can theoretically be anything (segfault, garbage value etc)

Thread 1

```
x.store(1, stdmo::relaxed);  // (a)
y.store(2, stdmo::release);  // (b)
x.store(3, stdmo::relaxed);  // (c)
z.store(4, stdmo::release);  // (d)
x.store(5, stdmo::relaxed);  // (e)
```

Thread 2

```
while (y.load(stdmo::acquire) != 2) { }  // (p)
cout << x.load(stdmo::relaxed);          // (q)
while (z.load(stdmo::acquire) != 4) { }  // (r)
cout << x.load(stdmo::relaxed);          // (s)
```

- The first cout can print 1,3,5. The secon cout can print 3,5. They will never print 0 due to the acquire release.

Fences

- Enforces memory order without modifying the data
- Can be used with Relaxed MO to enforce ordering while preserving concurrency
- **Memory Barrier** puts a line that certain operations cannot pass
- **Atomic fence** with acquire release MO prevents all preceding read and writes from moving past all subsequent stores

T3: Debugging Concurrent C++ Programs

Protecting shared resource with unique_ptr

- std::unique_ptr is a smart pointer that owns and manages another object through a pointer and disposes of that object when the unique_ptr goes out of scope.
- Destroys object with the provided $deleter$

```
{
std::unique_ptr<int> foo =
    std::make_unique<int>(5);
/* custom deleter for File */
auto deleter = [](FILE* f) { fclose(f); };
auto bar = std::unique_ptr<FILE,
    decltype(deleter)>(fopen("file.txt",
    "w"), deleter);
}
/* Foo and bar are destroyed here */
```

Protecting against shared lifetimes

- std::shared_ptr is a smart pointer that retains shared ownership of an object through a pointer.
- When shared ptr is copied, we increase the count by 1. When it is destroyed, we decrease the count by 1. When count is 0, we delete the object
- While deleting shared ptr is thread safe (since it just deceremnts the counter), the object wrapped by shared ptr may not be (data races can still happen)

Possible bugs with shared_ptr

- Unsynchronised access to managed object - WR, WW can still happen if exclusive lock is not used
- Data race on ptr (UAF): Overwrites to ptr deletes the old value and make a new one. But the reader may still be using the old ptr. Do not pass shared ptr by reference.
- Circular reference: A points to B, B points to A. Both will never be deleted since the reference count will be at least
  1. E.g. A.prev = B, B.next = A.
     - Using DLLNode* prev and DLLNode next. Only the next pointer copies the shared pointer, so there is no cycle.

Implementing shared ptr

- The semantics of a shared ptr is that it manages when to destroy a shared resource.
- This means that we need to keep a shared count, that is stored as a reference
- Since this is shared, we need to use a mutex (see 3.4) or atomic operations (see 3.5) to protect it
- During deletion (count == 0), we cannot delete the mutex in the critical section since the std::unique lock will call unlock when exiting the CS
- So we store the decremented value of counter in a temp var and check $temp == 0$ outside of the critical section
- This is correct since $temp == 0$ implies no thread can modify the counter so no data race
- Note: Use an atomic int for the atomic implementation

Extra overhead of shared pointer

- Ambiguous ownership: Most resources should have a single owner using std::unique ptr and use a singleton pattern
- Memory leaks: Cyclic references, as shown above, can cause memory leaks. Std::unique ptr does not face this.
- Performance overhead: Maintaining reference counts result in unnecessary synchronisation. Initialising memory on the heap instead of the stack (since shared ptr is shared) may slow down the program and reduce cache locality

T4 - lock free programming

This tutorial implements the threadsafe queue in L5 using atomics.

Lock free

> If multiple threads are operating on a DS, then after a bounded number of steps one of them must complete its operation

Problems faced by producers

- Producers may data race by overwritting each other's job

1. Use queue_back.exchange(new dummy), which returns the current work node to the producer to edit while setting the back to be a new dummy node (contains null)

- A producer may not be synchronised with consumers, causing consumers to read an invalid state

1. Use acquire release to update the $next$ pointers

Problems faced by consumers

1. Consumer may race wuth each other (by consuming a node twice, consuming another consumer's node, deleting the node (UAF))
   - Doing queue_front.exchange(queue_front.next) is not good enough as ew only want to exchange if the current node does not contain null
   - A **CAS** is needed
2. Consumers may not be synced with producers
   - Use acquire release to update the $next$ pointers

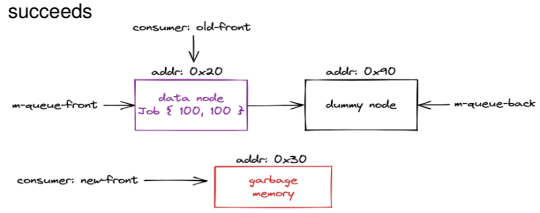
CAS pattern

- Performs a comparison on the memory location, checking if the current value is the same as the expected value. Exchange if true
- If the side effect is not monotonic, we might face **ABA** problem
- **Weak CAS** is conducted in a sequence of instructions. It is possible for it to fail spuriously. It is cheaper and should be used if CAS is used in a loop and each loop is cheap.
- **Strong CAS** is conducted in a single $cmpxchng$ instruction. It is more expensive but will not fail spuriously.

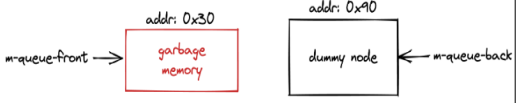
```
/* Ensures that the value read gets more
    updated with each loop */
Node* old_front_next = old_front.node ->
    next.load(stdmo::acquire);
...
/* Can be relaxed because of the previous
    acuire */
m_queue_front.compare_exchange_weak(old_front
    old_front_next,
    std::memory_order_relaxed)
```

ABA problem

- The next pointer in a Node is obviously non-monotonic (and so are the values stored in the nodes)
- It is possible for an old front to be consumed, and then re-inserted with the same value (but not necessarily at the front). A thread that read the old front before it was consumed will not be able to detect this, and CAS succeeds



- But the new old front may not contain the same next pointer as the value read by the thread from the old old front.



ABA: final broken state

Solving ABA with generation-counted pointers
- Include a generation counter alongside the data we are performing CAS on to differentiate between versions

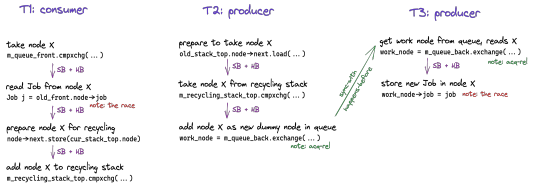
```
struct alignas(16) GenNodePtr
{
    Node* node;
    uintptr_t gen;
};

static_assert(
std::atomic<GenNodePtr>::is_always_lock_free
);

alignas(64) std::atomic<Node*>
    m_queue_back;          // producer end
alignas(64) std::atomic<GenNodePtr>
    m_queue_front; // consumer end
```

- When we initialise the new front, we set its gen to be old front.gen + 1.
- (Of course, overflow is possible so it is not fool proof)

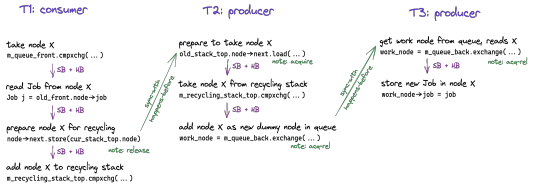
UAF
- gen ptr ensures that CAS fails when other consumers have changed the queue, but it is possible to dereference a pointer froma possible stale value of m_queue_front
- Then, that node may be allocated again on the same address. And the constructor of std::atomic will raise with the load()
- **Solutions:**
1. Never free anything (let the memory leak)
2. Mark nodes for deletion, and delay the deletion when the entire queue is free (no other threads acccessing queue)
3. Use reference counting (atomic::shared ptr) to know when there are no more remaining references. This is not lock free.
4. Hazard pointers to track which threads have references to which objects

Recycling center
- Suppose we go with solution 2, we can set up a recycling center that maintains a concurrent stack of free nodes (see 4.1 for implementation details) so we can reuse deleted nodes.
- Push will obtain the node from the recycling center if it is not empty, or allocate a new node if it is empty
- Since nothing is freed, we cannot get UAF

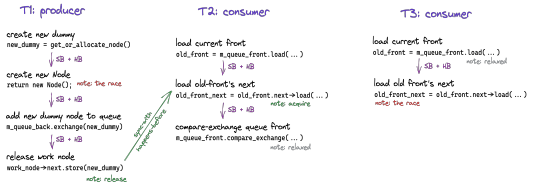
Data race in recycling stack



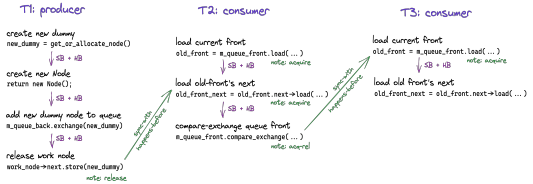
- The race happens because consumer tries to read from the work_node before the producer has written to it
- This is possible since htere is no synchronisation between T1 and T3
- To fix this, we can make the load store of node.next acquire release such that T1 and T2 (by transitive T3 as well) are in sync



Internal Data Race



- By allowing consumers to overtake producers threads (use yield), we can have a seemingly-impossible data race
- The fix is to slap acquire release on the load store of m_queue_front



Benchmarking
- Some common metrices are: CPU cycles, real clock time, MFLOPs (floating point operations per second)
- There are 4 possible set ups to test for
1. Single Producer, Single Consumer (SPSC)
2. Single Producer, Multiple Consumers (SPMC)
3. Multiple Producers, Single Consumer (MPSC)
4. Multiple Producers, Multiple Consumers (MPMC)
- Use barier to ensure that all producers and consumer threads arrive at the the start of the benchmark before starting the timer and commencing the execution

T5: Goroutines and channels

Silly mistakes
- Not using wait group or waiting for channels to finish and ”join”
- Allowing data race by not granting exclusive access by passing objects using channels s.t only 1 owner exists at any point in time
- Creating a deadlock using cyclic dependencies between channels
- Using buffered channel to resolve deadlock - bad idea since there are now multiple owners to values and a buffered channel becomes effectively unbuffered when it is full + more interleavings possible
- Closing the channel too early and resulting in a panic
- Not closing the channel and never stops reading

MPSC Queue using Channels
- For associative and identity operations, we can simply have the producers update a local copy from 0 and send it back to the consumer once done
- This can be done either by having a channel of channels with one channel for each producer (1.2), or by having a single channel and using a for loop to read exactly N times from the channel (1.1).

MPMC Queue using Channels
- The default MPMC GO provides contains mutex so it is not lock free
- We can implement a non-blocking, lock-free MPMC queue using the select-default idiom (See 2.2) (Note: default is a nexist for when all cases are blocked)

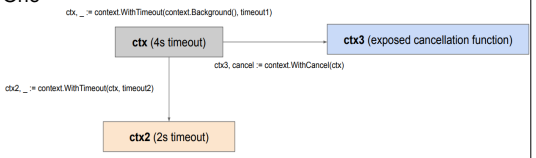
context.Context
- To canccel an Asynchronous task, we can use context.Context

```
func producer(ctx context.Context, q
    queue) {
    for {
        select {
        case q <- 1: // normal enqueue to
            q with type alias to chan int
        case <-ctx.Done():
            return
        }
    }
}
```

- Context can be set up with a timeout ctx := context.WithTimeout(context.Background(), 1*time.Second)

T6 Advanced GO Concurrency Patterns

Exit Conditions
- Manage different exit conditions for different goroutines and share data across different goroutines
- Use a context tree to combine multiple exit conditions into One



- Sample:

```
func main() {
    ctx, cancel :=
        context.WithCancel(context.Background())
    ctx3, cancel :=
        context.WithCancel(ctx)
    defer cancel()

    go func() {
        <-ctx3.Done()
        fmt.Println("Context cancelled")
    }()
...
}
```

- It is possible for ctx 3 to not be seen as cancelled if ctx (main) is cancelled first. So GO can kill the thread ctx3 is on without printing
- We can use waitgroup or done channel to ensure that main only terminates when all its threads (including ctx3) are done

Fan-Out, Fan-In
- Fan-out distributes intense workloads across multiple workers
- Fan-in centralises results but this is not necesarily serialised
- We can serialise it using higher-order channels that uses an unbuffered channel, but this also greatly limits concurrency

Pipelining
- Useful for resource constrained parallelism while maintaining a separation of constraints
- E.g for a 3 stage program that is CPU intensive at stage 2 (bottleneck) and can only run 10 threads concurrently at stage 3 (due to memory constraints), we can pipeline multiple threads at stage 2 and only 10 threads at stage 3 to maximise throughput

T7 - Classic concurrency problems in C++ and Go

H2O
- Sepcificaiton: 2H + O $\rightarrow$ H2O, only the atoms involved can execute concurrently

H2O - Solution
- Naive: Use a barrier(3) - this fails as there’s no guarantee that we will have 3 hydrogens
- GO Daemon:
1. Create a private communication channel
2. (Precommit) send channel in an arrival request to the Daemon
3. (Commit) receive the signal to proceed from the daemon
4. (Postcommit) send the signal that bonding is done to the daemon
- The issue with using a daemon is that the daemon is a single point of failure and it leaks memory unless we shut down the daemon with a context
- GO leader goroutine:
1. (follower behaves the same way as the daemon)
2. Become leader by locking mutex
3. (Precommit) Recieve arrival request from follower(s)
4. (Commit) Tell the hydrogen atoms to proceed and begin bonding
5. Bond
6. (postcommit) wait for the hydrogen atoms to finish bonding

7. Step down as the leader by unlocking the mutex
- The single point of failure issue is resolved since the mutex channel is closed when the leader exits

FIFO Semaphore

Unfortunately, both semaphores in C++ and blocking queues in Go are not guaranteed to be FIFO. This can lead to starvation.

C++: Ticket Queue
- We assign a ticket to arrivals and keep 2 atomics next_ticket (next to give to arrivals) and now_serving (next to serve)
- In acquire, the arrivals will wait (on a spin lock or conditional variable) till now serving ge my ticket.
- This is guaranteed to release arrivals in FIFO order as long as acquire and release are paired
- Note: Release should both increment the now_serving with fetch add and **notify_all** arrivals, this will cause spurious wakes but **notify_one** may notify the wrong arrival and cause starvation (no progress can be made)

C++: Using a queue of semaphores
- Have a counter to track the number of blocked threads
- Have a queue of semaphores to store the blocked threads
- Counter is initialised to be the value of the semaphore
- Acquire(): If counter is positive, we decrement it and return without blocking. Else, we add waiter to queue and block on its own semaphore
- Release(): If the queue is empty, we do nothing and increment the counter. Else, we pop the first waiter and release its semaphore.
- In both cases, we do not change counter if we are popping or pushing into queue
- **Problem:** there are roughly 2 allocations needed every time a new waiter is added to the queue, this is quite expensive and possibly slow to do on heap.

Go: Using a buffered channel
- Use a buffered channel the size of capacity and load it with (capacity - initial size) number of elements
- Sends to the channel decrements the number of elements and receives increments it
- When there is no more free space, subsequent sends will block
- This is practically FIFO but not guarateed by Go specification

Go: Using a daemon goroutine
- We can use a daemon thread to manage the blocked threads, just like the queue in the C++ implementation
- Technically an acquire can still block on the first send to s.acquire channel if the acquire channel is full (and we go back to the issue of not being FIFO)

T8: Safety and Concurrency in Rust

Lifetime
- 'x if a lifetime specifier and the 'static life time is special in the sense that those with a static lifetime lives for as long as the prorgram. And any data captured by the closure must live for at least as long as this.

Shared memory in rust
- It is possible to do share mutable references using Arc(mutex(...))
- A new Arc is created each time we clone the arc to pass to the new thread
- To prevent dangling pointers (UAF), we need to join all threads that takes an Arc in its closure
- The mutex lives on heap, which is potentially slower than a stack access

Lock Poisening
- It is possible for a thread to be aborted while holding a lock. This causes the data protected by the lock to be in an inconsistent state.
- Other threads accessing the same mutex should unwrap before use so that the error propagates

Scoped Thread
- Scoped thread has an implicit join at the end of it. So we can pass shared references of our mutex to threads in the scoped thread without risking UAF
- Arc is not needed and the mutex can stay on the stack

Interior mutability
- Library codes can contain constructs (e.g. mutex) that wraps around a mutable reference to a value to allow the passing of multiple mutable references
- The authors of such code is responsible to ensure safety
- For users, data races can only arise from the library or the compiler as long as they do not use unsafe elsewhere in their code

Classical Synchronisation Problems

Problem	CS problem
Barrier	Wait until threads/processes reach a specific point in the execution
Producer-consumer	Model interactions between a processor and devices that interact through FIFO channels.
Readers-writers	Model access to shared memory
Dining philosophers	Allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
Barbershop	Coordinating the execution of a processor.
FIFO Semaphore	Needed to avoid starvation and increase fairness in the system.
H2O	Allocation of specific resource to a process.
Cigarette smokers	The agent represents an operating system that allocates resources, and the smokers represent applications that need resources.

Comparison between Rust, Go, C++

Ownership

- C++ has RAII to manage resources, moveable but not copyable reference

Testing and debugging

- C++ cannot catch many race conditions and concurrency bugs at compile time, and requires testing at run time
- Tsan and Asan has verbose output (unlike rust compiler)
- Catching concurrent bugs at run time runs into heisenbugs

Memory Management: GO vs Rust

- Go has a garbage collector, Rust has a manual memory management system based on ownership and borrowing - need to be more careful with rust
- Rust's borrow checker does guarante lifetime and exclusive access at compile time

Rust Concurrency Model

- Rust's concurrency is provably correct - the compiler can catch thread safety bugs at compile time. This eliminates basic concurrency bugs

Rust Vs C++ Memory model - thanks chatgpt

In Rust, ownership is a core concept of the language, and moving ownership is the default behavior when you pass a value to a function or assign it to another variable. When ownership is moved, the original variable is no longer valid, and trying to use it will result in a compile-time error. This is a safety feature designed to prevent problems like double-free, use-after-free, and data races.

In C++, the copy constructor is called when an object is initialized from another object of the same type. When you pass an object by value to a function or return an object from a function, the copy constructor is used to create a new copy of the object. This means that, unlike Rust, C++ makes a copy of the object by default, and the original object remains valid.

The difference between Rust's ownership and C++'s copy constructor is that Rust enforces strict ownership rules at compile time, which helps prevent memory-related bugs, while C++ relies on the programmer to manage memory and resources correctly. This can lead to issues like memory leaks, double-free errors, and other memory-related bugs if not handled correctly in C++.

To have similar behavior to Rust's ownership model in C++, you can use move semantics, introduced in C++11, with move constructors and std::move. This allows you to transfer ownership of a resource to another object, leaving the original object in a valid but unspecified state. However, using move semantics in C++ is still a matter of choice and requires a clear understanding of when and how to use them, whereas Rust enforces strict ownership rules by default.

In summary, Rust's ownership system enforces strict rules that help prevent memory-related bugs, while C++ provides flexibility with copy constructors and move semantics. Rust's ownership model is more restrictive and enforced at compile-time, while C++ relies more on the programmer's ability to manage memory and resources correctly.

Random Trivia

Atomic shared ptr but not atomic unique ptr

- std::shared_ptr needs to allocate a control-block where the strong and weak count are kept, so type-erasure of the deleter came at a trivial cost (a simply slightly larger control-block).
- Note: Atmoic shared_ptr is not lock free
- std::unique_ptr has a zero-overhead policy; using a std::unique_ptr should not incur any overhead compared to using a raw pointer.
- But the deleter of a std::unique_ptr may not be zero cost (that is, it may stores some state)
- Since there is no guarantee that we can do CAS on the deleter without the use of mutex (which comes with extra overhead), we cannot do CAS on the unique ptr
- Another argument is that implementing an atomic unique ptr requires weakening the precondition on the atomic template parameter to avoid deadlocks. And that there is no effective mechanism to test for the weaker property