CS3211 Summary Sheet

L1 – Introduction to Concurrency

Concurrency refers to two or more separate activities happening at the same time. For computers, concurrency refers to a single system performing multiple independent activities in parallel, rather than sequentially (multitasking in OS).

Why study concurrency?: Traditionally concurrency was achieved through task switching. The increased prevalence of computers that can genuinely run multiple tasks in parallel rather than giving the illusion of doing so. (illusion of concurrency vs true concurrency)

Concurrency vs Parallelism

Concurrency

- Two or more tasks that can start, run, and complete in overlapping time periods
- They might not be running (executing on CPU) at the same instant
- Two or more execution flows make progress at the same time by interleaving their executions or by executing instructions (on CPU) at exactly the same time

Parallelism

- Two or more tasks can run (execute) simultaneously, at the exact same time
- Tasks do not only make progress, but they also actually execute simultaneously

Hardware enables true concurrency.

Computers are genuinely able to run more than one task in parallel

- Multicore processors are used everywhere
- Multiple processors used everywhere
- High performance computing

Hardware threads dictate the amount of true concurrency.

- Processors can have multiple cores
- A core can support multiple hardware threads (SMT - simultaneous multithreading)

Enabling concurrency:

- Multiple processes vs multiple threads (safety vs performance)

Process Interaction with OS

Exceptions

- Executing a machine level instruction can cause exception
- For example: Overflow, Underflow, Division by Zero, Illegal memory address, Mis-aligned memory access
- Synchronous: Occur due to program execution
- Have to execute an exception handler

Interrupts

- External events can interrupt the execution of a program
- Usually hardware related: Timer, Mouse Movement, Keyboard Pressed etc.
- Asynchronous: Occur independently of program execution
- Have to execute an interrupt handler

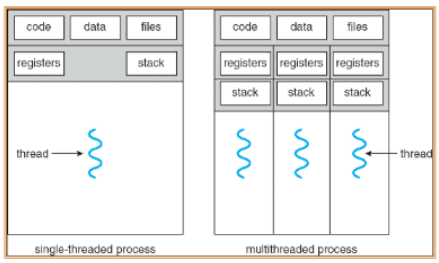
Disadvantages of processes

- Creating a new process is costly
  - Overhead of system calls
  - All data structures must be allocated, initialized and copied
- Communicating between processes costly
  - Communication enabled by the OS

Threads

- Extension of process model:
  - A process may consist of multiple independent control flows called threads
  - The thread defines a sequential execution stream within a process(PC, SP, registers)
- Threads share the address space of the process
- Thread generation is faster than process generation
  - No copy of the address space is necessary
- Different threads of a process can be assigned run on different cores of a multicore processor

2 types of threads: User-level threads & Kernel threads



Race condition (both condition must hold true)

1. Two concurrent threads (or processes) access a shared resource without any synchronization
2. At least one thread modifies the shared resource

- Solution: control access to these shared resources
- Necessary to synchronize access to any shared data structure
  - Buffers, queues, lists, hash tables, etc

Mutual exclusion

- Use **mutual exclusion** to synchronize access to shared resources
  - This allows us to have large atomic blocks
- Code sequence that uses mutual exclusion is called **critical section**
  - Only one thread at a time can execute in the critical section
  - All other threads have to wait on entry
  - When a thread leaves a critical section, another can enter

Critical Section Requirements

1. **Mutual Exclusion (Mutex):** If one thread is in the critical section, then no other is
2. **Progress:** If some thread T Is not in the critical section, then T cannot prevent some other thread S from entering the critical section
3. **Bounded Waiting** (no starvation): If some thread T is waiting on the critical section, then T will eventually enter the critical section
4. **Performance**: The overhead of entering and exiting the critical section is small w.r.t. the work being done within it

Critical section requirements – details

- Requirements:
  - **Safety property**: nothing bad happens
  - **Liveness property**: something good happens (Progress, Bounded Waiting)
  - **Performance requirement**
- Properties hold for each run, while performance depends on all the runs
- Rule of thumb: When designing a concurrent algorithm, **worry about safety first (but don't forget liveness!)**

Mechanisms

- **Locks**: Primitive, minimal semantics, used to build others
- **Semaphores**: Basic, easy to get the hang of, but hard to program with
- **Monitors**: High-level, requires language support, operations implicit
- **Messages**: Simple model of communication and synchronization based on atomic transfer of data across a channel. Direct application to distributed systems. Messages for synchronization are straightforward (once we see how the others work).

Deadlock

Deadlock exists occurs when the waiting process is still holding on to another resource that the first needs before it can finish.
Deadlock is a problem that can arise: when processes compete for access to limited resources & processes are incorrectly synchronized.

Condition for deadlock

Deadlock can exist **if and only if** the following four conditions hold simultaneously:

1. **Mutual exclusion** – At least one resource must be held in a non sharable mode
2. **Hold and wait** – There must be one process holding one resource and waiting for another resource
3. **No preemption** – Resources cannot be preempted (critical sections cannot be aborted externally)
4. **Circular wait** – There must exist a set of processes [P1, P2, P3,...,Pn] such that P1 is waiting for P2, P2 for P3, etc.

Dealing with deadlock

- There are four approaches for dealing with deadlock:
- Ignore it–how lucky do you feel?
- Prevention–make it impossible for deadlock to happen
- Avoidance–control allocation of resources
- Detection and Recovery–look for a cycle in dependencies

Starvation

Starvation is a situation where a process is prevented from making progress because some other process has the resource it requires
Starvation is a side effect of the scheduling algorithm.

- OS: A high priority process always prevents a low priority process from running on the CPU
- One thread always beats another when acquiring a lock

Disadvantages of concurrency

- Concurrency issues
- Maintenance difficulties
  - Complicated code
  - Debugging is challenging
- Threading overhead
  - Stack
  - Context switching

Concurrency benefits

- Separation of concerns
- Performance
  - Take advantage of the hardware
  - Optimization strategy

Types of parallelism

- **Task parallelism**: Do the same work faster
- **Data parallelism**
  - Embarrassingly parallel algorithms
  - Do more work in the same amount of time

Concurrent and Parallel Programming Challenges
- Finding enough parallelism (Amdahl's Law!)
- Granularity of tasks
- Locality
- Coordination and synchronization
- Debugging
- Performance monitoring

Execution of a (concurrent) program (C/C++)
- Compilation and linking: Done by the compiler
- Loading: The loader is usually specific to the OS
- Execution: Coordinated by the OS. The program gets access to CPU, memories, devices, etc.

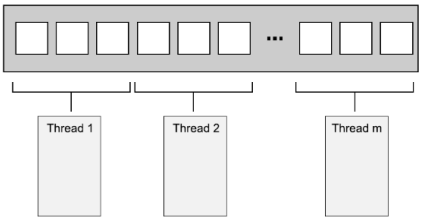
Building flow in C/C++
- Preprocessor: Replaces preprocessor directives (#include and #define)
- Compiler: Parses pure C++ source code and converts it into assembly
- Assembler: Assembles that code into machine code
- Linker: Produces the final compilation output from the object files the compiler produced

L2 – Tasks, Threads and Synchronization (in Modern C++)

Program Execution
- **Decomposition** of computations
- **Assigning** tasks to threads
- **Orchestration**
  - Structuring communication
  - Adding synchronization to preserve dependencies
  - Organizing data structures in memory
  - Scheduling tasks
- **Mapping** of threads to physical cores

Distributing work to threads – Task Parallelism
- Divide the work into tasks to make the threads specialists
  - Same types of tasks assigned to the same thread (aka pipeline)
  - Divide the work by task type to separate concerns
- Dividing a sequence of tasks among threads to achieve a complex solution
  - Pipeline: each thread is responsible for a stage of a pipeline
- Use task pools and the number of threads to serve the task pool

Distributing work to threads – data parallelism
- Divide data



C++ history
- 1998 –the original C++ standard published
  - No support for multithreading!
  - Use external libraries to manage threads in your C/C++ programs: Pthread
- 2011 – C++11 (or C++0x) standard published
  - new "train model" of releases
  - support of multithreaded programs

C++98
- Does not acknowledge the existence of threads
- Effects of language elements assume a sequential abstract machine
  - No memory model
- Multithreading was dependent on **compiler-specific extensions**
  - C APIs, such as POSIX C standard and Microsoft Windows API, used in C++
- Very few formal multithreading-aware memory models provided by compiler vendors
- Application frameworks, such as Boost and ACE, wrap the underlying platform-specific APIs
  - Provide higher-level facilities for multithreading

C++11 multithreading
- Write portable multithreaded code with **guaranteed behavior**
  - Multithreading without relying on platform-specific extensions
- **Thread-aware memory model**
- Includes classes for managing threads, protecting shared data, synchronization between threads, low-level atomic operations
- Use of concurrency to improve **application performance**
  - Take advantage of the increased computing power
  - Low abstraction penalty - C++ classes wrap low-level facilities
  - Low-level facilities: atomic operations library

Managing threads
- Every program has at least one thread: Started by the C++ runtime & runs *main()*
- Use **std::thread** to add threads
- Identify a thread using get_id()

1-2: Thread with a function
6-16: Thread with a function object (callable type)
24-25: Thread with a function object
30-33: Thread with a lambda expression (local function instead of a callable object)

```
1  void do_some_work();
2  std::thread my_thread(do_some_work);
```

```
6  class background_task
7  {
8  public:
9      void operator()() const
10     {
11         do_something();
12         do_something_else();
13     }
14 };
15 background_task f;
16 std::thread my_thread(f);
```

```
20 std::thread my_thread(background_task());
```

```
24 std::thread my_thread((background_task()));
25 std::thread my_thread{background_task()};
```

```
30 std::thread my_thread([]{
31     do_something();
32     do_something_else();
33 });
```

Note: Line 20 declares a function that returns a thread. It declares a my_thread function that takes a single parameter (of type function-taking-no-parameters-and-returning-a-background_task-object) and returns a std::thread object. It does not launch a new thread! Use curly brackets like in line 24-25 to treat it as a function object.

Wait or detach?
- Wait for a thread to finish
  - Use join() on the thread instance, exactly once
  - Make sure to join the thread even when there is an exception! (try-catch)
  - Use joinable() to check
- Detach the thread
  - Use detach()
  - Extra care is needed with local variables passed to the thread
  - Local variables passed as parameters to the thread function might end their lifetime before the thread ends

```
void oops()                                   local variable
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread my_thread(my_func);
    my_thread.detach();
}
```

Passing arguments to a thread function

- Lines 1-2: Passing a function and arguments by value

```
1  void f(int i,std::string const& s);
2  std::thread t(f,3,"hello");
```

- Lines 11-18: oops might exit before the `buffer` is converted to `std::string` within the new thread (passing a reference to buffer)

```
11 void f(int i,std::string const& s);
12 void oops(int some_param)
13 {
14     char buffer[1024];
15     sprintf(buffer, "%i",some_param);
16     std::thread t(f,3,buffer);
17     t.detach();
18 }        buffer out of scope
```

- Lines 21-28: the conversion happens before passing the argument to the thread through an explicit cast

```
21 void f(int i,std::string const& s);
22 void not_oops(int some_param)
23 {
24     char buffer[1024];
25     sprintf(buffer,"%i",some_param);
26     std::thread t(f,3,std::string(buffer));
27     t.detach();          Make sure buffer not used in 't'
28 }
```

Passing arguments by reference

- Lines 31-39: data is passed by value (copy)

```
31 void update_data_for_widget(widget_id w,widget_data& data);
32 void oops_again(widget_id w)
33 {
34     widget_data data;
35     std::thread t(update_data_for_widget,w,data);
36     display_status();
37     t.join();
38     process_widget_data(data);    data not modified
39 }
```

- Line 41: Wrap the arguments in std::ref

```
41 std::thread t(update_data_for_widget,w,std::ref(data));
```

Ownership in C++
- An **owner** is an object containing a pointer to an object allocated by new for which a delete is required
- Every object on the free store (heap, dynamic store) must have exactly **one** owner.
- C++'s model of resource management is based on the use of constructors and destructors
  - For scoped objects, destruction is implicit at scope exit
  - For objects placed in the free store (heap, dynamic memory) using new, delete is required
- Objects can also be allocated using malloc() and deallocated using free() (or similar functions), but the techniques described for new and delete apply to those also

RAII - Resource Acquisition Is Initialization
- C++ programming technique
- Binds the life cycle of a resource that must be acquired before use (allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection—anything that exists in limited supply) to the **lifetime** of an object.

Lifetime

- The **lifetime of an object begins** when:
  - storage with the proper alignment and size for its type is obtained, and
  - its initialization (if any) is complete (including default initialization via no constructor or trivial default constructor) (except that of union and allocations by std::allocator::allocate)
- The **lifetime of an object ends** when:
  - if it is of a non-class type, the object is destroyed (maybe via a pseudo-destructor call), or
  - if it is of a class type, the destructor call starts, or
  - the storage which the object occupies is released, or is reused by an object that is not nested within it.
- Lifetime of an object is equal to or is nested within the lifetime of its storage, see storage duration.
- The **lifetime of a reference begins** when its initialization is complete and **ends** as if it were a scalar object. Note: the lifetime of the referred object may end before the end of the lifetime of the reference, which makes **dangling references** possible.

All objects in a program have one of the following storage durations:

- **automatic** storage duration. The storage for the object is allocated at the beginning of the enclosing code block and deallocated at the end. All local objects have this storage duration, except those declared static, extern or thread_local.
- **static** storage duration. The storage for the object is allocated when the program begins and deallocated when the program ends. Only one instance of the object exists. All objects declared at namespace scope (including global namespace) have this storage duration, plus those declared with static or extern.
- **thread** storage duration. The storage for the object is allocated when the thread begins and deallocated when the thread ends. Each thread has its own instance of the object. Only objects declared thread_local have this storage duration. thread_local can appear together with static or extern to adjust linkage.
- **dynamic** storage duration. The storage for the object is allocated and deallocated upon request by using dynamic memory allocation functions.

Ownership of a thread

- std::thread instances own a resource: Manage a thread of execution
- Instances of std::thread are
  - Movable (transfer ownership)
  - are not copyable
- (Same ownership semantics as std::unique_ptr)

```
1  void some_function();
2  void some_other_function();
3  std::thread t1(some_function);
4  std::thread t2=std::move(t1);
5  t1=std::thread(some_other_function);
6  std::thread t3;
7  t3=std::move(t2);
8  t1=std::move(t3);
```

ERROR! Compiler won't allow

can detach t1 before line 8 to fix

L4: Move t1 into t2, t1 doesn't ref anything anymore. t2 now refering to some_function.
L5: t1 refers to a new thread running some_other_function
L7: t3 gets thread that is referred by t2 (some_function)
L8: ERROR: compiler won't allow.

Transferring ownership of a thread

- Lines 11-19: Transfer ownership out of function
- Lines 21-28: Transfer ownership into a function

```
11 std::thread f() {
12     void some_function();
13     return std::thread(some_function);
14 }
15 std::thread g() {
16     void some_other_function(int);
17     std::thread t(some_other_function,42);
18     return t;
19 }
```

```
21 void f(std::thread t);
22 void g()
23 {
24     void some_function();
25     f(std::thread(some_function));
26     std::thread t(some_function);
27     f(std::move(t));
28 }
```

cannot just call f(t) - threads not copyable

Synchronizing multiple threads

1. Concurrent access to shared data: Mutex
2. Concurrent actions: Condition variable, Monitor

Synchronizing concurrent accesses

- If all shared data is **read-only** – no problem
  - the data read by one thread is unaffected by another thread is reading the same data
- Modifying shared data comes with many challenges

Invariants

Invariants – statements that are always true about a particular data structure

- Often broken during an update on the data structure
- Example: this variable no_of_items contains the number of items in the list

Use invariants to reason about program correctness

Delete a node from a doubly linked list

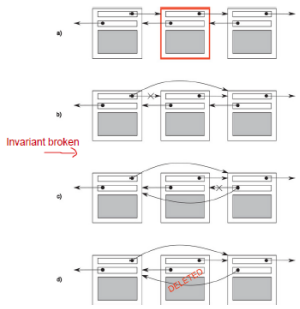
*Invariant is broken during the delete

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.



Problems with sharing data among threads

- Programs must be designed to ensure that changes to a chosen data structure are correctly synchronized among threads
  - Data structures are immutable (data never changes), or
  - Protect data using some locking: external or internal
- Invariants are often broken during an **update** on the data structure
  - Other threads might work with data while the invariant is broken
- Race conditions

Race conditions vs. data races

Race condition

- The outcome depends on the relative ordering of execution of operations on two or more threads
- The threads race to perform their respective operations
- Usually, race condition is a flaw that occurs when the timing or ordering of events affects a program's correctness.
- Might not always be bad – e.g. if orderig is not importat

Data race

- A data race happens when there are two memory accesses in a program where both:
  - target the same location
  - are performed concurrently by two threads
  - are not reads
  - are not synchronization operations
- Causes **undefined behavior**

Avoiding race conditions

- Chances of the problematic execution sequence occurring increases
  - high load in the system
  - the operation is performed more time
- Simplest option: wrap your data structure with a protection mechanism
  - Ensure that only the thread performing a modification can see the intermediate states while the invariants are broken
  - C++ provides such mechanisms for locking

Synchronization primitive: mutex

- Mutex = mutual exclusion
- Provides serialization
  - Threads take turns accessing the data protected by the mutex

Mutex in C++

Not recommended usage:
1. Construct an instance of std::mutex,
2. Lock it with a call to the lock() member function
3. Unlock it with a call to the unlock() member function
- Must **remember** to call unlock() on every code path out of a function, including those due to **exceptions**!

Recommended usage: use **std::lock_guard** class template

- Implements RAII idiom for a mutex
- Locks the supplied mutex on construction and unlocks it on destruction

In C++17: template argument deduction enables omitting the template argument list (line 18 instead of lines 8 and 13)

```
18  std::lock_guard guard(some_mutex);
```

Might think it is a function declaration, but it is not!! → Use curly brackets

Attempt on improving the usage of mutex

- Common to group the mutex and the protected data together in a class rather than use global variables
- Encapsulate the functionality and enforce the protection

Issues when passing references

- The call to the user-supplied funcfunction means that foo can pass in malicious_function to bypass the protection and then call do_something() without the mutex being locked
- **Don't pass pointers and references to protected data outside the scope of the lock**, whether by returning them from a function, storing them in externally visible memory, or passing them as arguments to user-supplied functions!

```
9  {
10 private:
11     some_data data;
12     std::mutex m;
13 public:
14     template<typename Function>
15     void process_data(Function func)
16     {
17         std::lock_guard<std::mutex> l(m);
18         func(data);
19     }
20 };
21 some_data* unprotected;
22 void malicious_function(some_data& protected_data)
23 {
24     unprotected=&protected_data;
25 }
26 data_wrapper x;
27 void foo()
28 {
29     x.process_data(malicious_function);
30     unprotected->do_something();
31 }
32
```

obtained reference to data - can be accessed without mutex later

can do_something() without mutex now!

Other types of locks

- **std::unique_lock** instance doesn't always own the mutex that it is associated with
  - Allows for locking the mutex later using std::defer_lock
- **std::lock()** function locks one or more mutexes at once without risk of deadlock
  - Use std::adopt_lock to indicate to std::lock_guard objects that the mutexes are already locked (should adopt the ownership of the existing lock on the mutex rather than attempt to lock the mutex in the constructor)
- **std::scoped_lock** instance accepts and locks a list of mutexes
  - Locks in the same way as lock()

Synchronizing multiple threads: Concurrent Actions

Waiting for an event or other condition

One thread is waiting for a second thread to complete a task

- Keep checking a flag in shared data (protected by a mutex) and have the second thread set the flag when it completes the task ~ wasteful
- Waiting thread sleeps for short periods between the checks using the std::this_thread::sleep_for()function ~ less wasteful
- Use **condition variable** to wait for an event to be triggered by another thread

Condition variable -definition

- A condition variable is associated with an event or other condition, and
- One or more threads can wait for that condition to be satisfied

How it works:

- When the condition is satisfied, the thread can then notify one or more of the threads waiting on the condition variable
- The notified threads wake up and continue processing

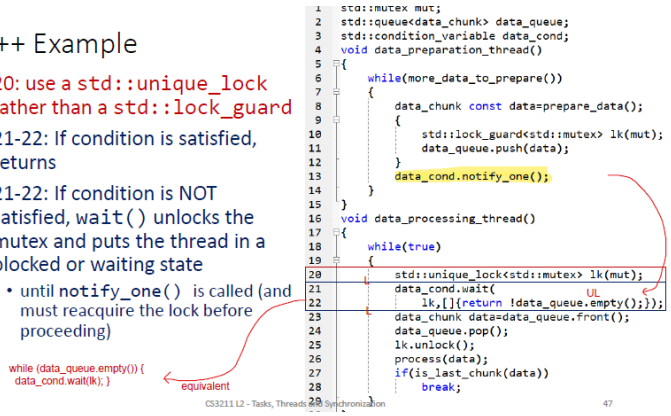
Two implementation in C++

std::condition_variable

- Works with std::mutex
- Simpler, lightweight, less overhead

std::condition_variable_any

- Works with anything mutex-like
- Potentially, additional costs in terms of size, performance, or OS resources



Condition variable behavior

During a call to wait(), a condition variable

- May check the supplied condition any number of times
  - Do not use a function with side effects for the condition check
- Checks the condition with the mutex locked
- Returns immediately if (and only if) the function provided to test the condition returns true

Spurious wake: The waiting thread reacquires the mutex and checks the condition, but not in direct response from a notification

Cond var: an optimization over a busy-wait

A basic, but inefficient, implementation

```
template<typename Predicate>
void minimal_wait(std::unique_lock<std::mutex>& lk,Predicate pred){
    while(!pred()){
        lk.unlock();
        lk.lock();
    }
}
```

There is no guarantee about how the condition variable is implemented. Programmers must be prepared for both: spurious wakes and waking only when notified.

C++ Concurrency In Action CH2, 3.1, 3.2, 4.1

Tutorial 1: Threads and Synchronization in C++:

```
int counter;
std::mutex mut;

void incr_ctr(int x) {
    mut.lock();
    counter += x;
    mut.unlock();
}

class increment_task {
    public:
// Overload operator(), like operator+, operator* ect.
// We are specifying what happens when an instance of the class is
called like a func: increment_task t; t();
        void operator()(int x) const {
            incr_ctr(x);
        }
};

int main() {
    //1. Thread with a function
    std::thread t0{incr_ctr, 5};
    std::thread t1(incr_ctr, 5);

    // 2. Thread with a function object (callable type)
    // A function object, or functor, is any type that
implements operator().
    increment_task task;
    std::thread t0{task, 2};
    std::thread t1(task, 2);

    // // 3. Thread with function object
    // // Note: both method works, but {} is preferred, note single
parantheses not allowed for first method
    std::thread t0((increment_task()), 4);
    std::thread t1{increment_task(), 3};  // <- Preferred

    //4. Thread with lambda expression (local function)
    std::thread t0{[]() { mut.lock(); ++counter; mut.unlock(); }};
    ...

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

Note:

- Mutexes can only be unlocked by the thread that locked it (c.f. binary semaphore where release/acquire can be called from different threads)

Why mutexes work: The standard argument is to first define a critical section that contains all accesses to the shared resource jobs, and argue that the std::mutex guarantees mutual exclusivity of threads in the critical section.

If we define the critical section to be the entirety of the enqueue and try_dequeue functions, we see that the calls mut.lock() and mut.unlock() do indeed bracket the critical section, and so only one thread may be in the critical section at any time. In particular, this guarantees that all accesses are synchronized, i.e. for any pair of accesses, one happens before the other.

Firstly, C++ formalizes what it means for the std::mutex to guarantee mutual exclusion. The ingredients are:

- The lock and unlock operations on a single mutex appears to occur in a **single total order**. (i.e. All threads observe the lock and unlock operations in the same order.)
- The lock operation blocks the thread until the thread owns the mutex, and the unlock operation gives up ownership of the mutex. Only one thread may own the mutex at a time, as seen by the single total order.
- The prior unlock operations *synchronize with* the lock operation.
-

Secondly, we examine the potential execution traces. Whenever jobs.push(job) and jobs.empty() run on different threads, they happen after their respective mut.lock()s, and happen before their respective mut.unlock()s.

Let's call the pair of lock/unlock operations surrounding jobs.push(job) "A", and the lock/unlock operations surrounding jobs.empty() "B". Since the lock and unlock operations occur in a single total order, we know that from the perspective of either thread, either:

1. "A" happens before "B" and "A"-unlock synchronizes with "B"-lock, or
2. "B" happens before "A" and "B"-unlock synchronizes with "A"-lock.

In case (1), jobs.push(job) happens before "A"-unlock, which synchronizes with "B"-lock, which happens before jobs.empty(). Thus jobs.push(job) happens before jobs.empty().

In case (2), jobs.empty() happens before "B"-unlock, which synchronizes with "A"-lock, which happens before jobs.push(job). Thus jobs.empty() happens before jobs.push(job). Either way, since one access must happen before the other, condition (b) holds, and therefore this is not a data race.

C++ Locks:

- std::lock_guard
  - Just lock & unlock in/out of scope
  - Cannot do manual lock/unlock
- std::scoped_lock (uses less memory than unique_lock)
  - Locks multiple locks without deadlock
  - Cannot do manual lock/unlock
  - Can unintentionally initialize it without a mutex
- std::unique_lock
  - Able to manually unlock, defer locking…

Problem 1: Mutex not unlocked when error happens . If jobs.push() throws an error (for example out of memory), the mut is not unlocked and the whole system will be locked.

```
void enqueue(Job job) {
  mut.lock();
  jobs.push(job);
  ...
```

Solution 1.1: Using try-catch

```
void enqueue(Job job) {
  mut.lock();
  try {
    jobs.push(job);
  } catch (...) {
    // unlock before bubbling up
    mut.unlock();
    throw;
  }
  mut.unlock();
}
```

Solution 1.2: Using unique_lock (or similar locks)

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

Concurrent Queue w/ Mutex

```
class JobQueue1 {
   std::queue<Job> jobs; std::mutex mut;
   public:
    JobQueue1() : jobs{}, mut{} {}

    void enqueue(Job job) {
        std::scoped_lock lock{mut};
        jobs.push(job);
    }

    std::optional<Job> try_dequeue() {
        std::scoped_lock lock{mut};
        try {
            if (jobs.empty()) {
                return std::nullopt;
            } else {
                Job job = jobs.front();
                jobs.pop();
                return job;
            }
        } catch (std::bad_alloc& a) {
            std::cout << "Caught a bad allocation\n";
            return std::nullopt;
        }
    }
};
```

Blocking Dequeue w/ Semaphore

- Ensure the locks are acquired in the same order!

```
class JobQueue3 {
    std::queue<Job> jobs; std::mutex mut; std::counting_semaphore<> count;
   public:
    JobQueue3() : jobs{}, mut{}, count{0} {}

    void enqueue(Job job) {
        std::scoped_lock l{mut};
        jobs.push(job);
        count.release();
    }

    std::optional<Job> try_dequeue() {
        // CORE SOLUTION DIFFERENCE
        // Try to decrement count, but return immediately if we cannot
        // count is decremented BEFORE the mutex is locked
        if (!count.try_acquire()) {
            return std::nullopt;
        }
        std::scoped_lock l{mut};
        Job job = jobs.front();
        jobs.pop();
        return job;
    }

    Job dequeue() {
        count.acquire();
        std::unique_lock lock{mut};
        Job job = jobs.front();
        jobs.pop();
        return job;
    }
};
```

Blocking Dequeue w/ Monitor

Components of a monitor are: mutex, condition variable, condition to wait for.

```
Job dequeue() {
    std::unique_lock lock{mut};
    while (jobs.empty()) { cond.wait(lock); }
    // Alternatively, this is exactly the same code
    // 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);
    }
    cond.notify_one();
}
```

cond.wait(lock) will atomically unlock the mutex and start waiting on the condition variable. When the condition variable is notified (or a spurious wakeup occurs), the lock will be reacquired before continuing.

We finish it off by having producers notify waiting consumer threads whenever a new element arrives.

The thread that intends to **modify** the shared variable must:

- Acquire a std::mutex (typically via std::lock_guard)
- Modify the shared variable while the lock is owned
- Call notify_one or notify_all on the std::condition_variable (can be done after releasing the lock)
-

Any thread that intends to wait on a std::condition_variable must:

- Acquire a **std::unique_lock<std::mutex>** on the mutex used to protect the shared variable
- Do one of the following:
  - Check the condition, in case it was already updated and notified
  - Call wait, wait_for, or wait_until on the std::condition_variable (atomically releases the mutex and suspends thread execution until the condition variable is notified, a timeout expires, or a spurious wakeup occurs, then atomically acquires the mutex before returning)
  - Check the condition and resume waiting if not satisfied

  or:
  - Use the predicated overload of wait, wait_for, and wait_until, which performs the same three steps

---

Lecture 3 –Atomics and Memory Model in Modern C++

Correctly synchronized program should behave as if:

- memory ops are actually executed in an order that appears equivalent to some sequentially consistent inter-leaved execution of the memory ops of each thread in your source code
- including that each write appears to be atomic and globally visible simultaneously to all processors.

Aggressive operation reordering is useful!

W→W: processor might reorder write operations in a write buffer (e.g., one is a cache miss while the other is a hit)
R→W, R→R: processor might reorder independent instructions in an instruction stream (out-of-order execution)
Valid optimizations when a program consists of a single instruction stream, but what about multiple threads?

- Languages need memory models
- Optimization **not visible** to programmer

As-if rule

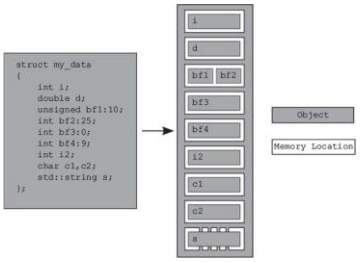
- The C++ compiler is permitted to perform any changes to the program as long as the following remains true:
  - Accesses (reads and writes) to volatile objects occur strictly according to the semantics of the expressions in which they occur. In particular, they are not reordered with respect to other volatile accesses on the same thread.
  - At program termination, data written to files is exactly as if the program was executed as written.
  - Prompting text which is sent to interactive devices will be shown before the program waits for input.
- Programs with undefined behavior are free from the as-if rule
  - Often these programs change observable behavior when recompiled with different optimization settings

Memory models provide a contract to programmers about how their memory operations will be reordered by the **compiler.**

You promise: to correctly synchronise your program (no race conditions)
System promise: to provide the illusion of executing the program you wrote

Multithreading-aware memory model (MM) in C++
- Programmers might not notice the MM
  - Using mutexes, futures, barriers do not require an understanding of the MM
- MM is essential to make all multithreading facilities work
- C++ is a systems programming language
  - Should not be needed to work with another lower-level language than C++
  - Allow programmers to get "close to the machine

Structure
- **Every variable is an object**, including those that are members of other objects.
- Every object occupies at least one memory location.
- Variables of fundamental types such as intor charoccupy exactly one memory location, whatever their size, even if they're adjacent or part of an array.
- Adjacent bit fields are part of the same memory location.



Concurrency
- Two threads access the same memory location
  - Read-only data doesn't need protection or synchronization
  - Either thread is modifying the data: potential for a race condition

Data Race - for two accesses to a single memory location from separate threads
- no enforced ordering between access
- one or both of those accesses is not atomic,
- and if one or both accesses is a write

→ Causes **undefined behavior**

Avoid data races: use critical section
- Code that must execute in isolation w.r.t.other program code

To implement critical section: Locks, Ordered atomics, Transactional memory

Undefined behavior
- The behavior of the complete application is now undefined, and it may do anything at all
- Serious bug and should be avoided at all costs
- To avoid undefined behavior: Understand modification orders

Modification order (MO)
- Composed of all writes to an object from all threads in the program
  - MO varies between runs
  - For every object in the program
- If the object is not one of the atomic types
  - Programmer is responsible for ensuring the threads agree on the modification order of each variable (through sufficient synchronization)
  - **Data race and undefined behavior**: different threads see **distinct sequences** of values for a single variable
- If atomic operations are used
  - Compiler is responsible for ensuring that the necessary synchronization is in place **(this does not mean "race-free")**

Modification order -requirements
- Once a thread has seen a particular entry in the modification order
  - Subsequent reads from that thread must return later values, and
  - Subsequent writes from that thread to that object must occur later in the modification order.
- A read of an object that follows a write to that object in the same thread must either return the value written or another value that occurs later in the modification order of that object
- All threads must agree on the modification orders of each individual object in a program
  - BUT they don't necessarily have to agree on the relative order of operations on separate objects

Atomic operations and types
- **Used to enforce modification order**
- Atomic operation is an indivisible operation
  - Always observed fully done from any thread in the system
- Atomic load and stores for an object
  - Load will retrieve either the initial value of the object or the value stored by one of the modifications
- (Non-atomic operation might be seen as half-done by another thread)
  - Non-atomic operation composed of atomic operations (for example, assignment to a struct with atomic members): other threads may observe mixed-up combination of completion of the operations

"atomic<> weapons" in C++
- Enable low-level synchronization
- Programmers need to understand and deal with the memory model
- Atomic types and operations on atomics
  - Low-level synchronization operations that reduce to 1-2 CPU instructions

Atomic types
- Found in the <atomic> header
- All operations on such types are atomic
- is_lock_free() member function returns true if operations on a given type are done directly with **atomic instructions**
  - X::is_always_lock_free is true if and only if the atomic type X is lock-free for all supported hardware that the current compilated code might run on
- Might emulate atomicity by using an internal mutex
  - The hoped-for performance gains will probably not materialize

Memory order in C++ (atomics)
- Each of the operations on the atomic types has an optional memory-ordering argument which is one of the values of the std::memory_order enumeration
- std::memory_order specifies how memory accesses, including regular, non-atomic memory accesses, are to be ordered around an atomic operation
- Reminder: with **no constraints** on a multi-core system:
  - Lack of modification order: when multiple threads simultaneously read and write to several variables, one thread can observe the values change in an order different from the order another thread wrote them
  - The apparent order of changes can even differ among multiple reader threads
  - Some similar effects can occur even on uniprocessor systems due to compiler transformations allowed by the memory model

std::memory_order enumeration
- **Default** ordering, the strongest ordering: **std::memory_order_seq_cst**
- **Store** operations can have **memory_order_relaxed**, **memory_order_release**, or **memory_order_seq_cst** ordering
- **Load** operations can have **memory_order_relaxed**, ~~memory_order_consume~~, **memory_order_acquire**, or **memory_order_seq_cst** ordering
- **Read-modify-write** operations can have **memory_order_relaxed**, ~~memory_order_consume~~, **memory_order_acquire**, **memory_order_release**, **memory_order_acq_rel**, or **memory_order_seq_cst** ordering

Sequentially consistent ordering
- Implies that the behavior of the program is consistent with a simple sequential view of the world
- **All threads must see the same order of operations**
- Operations can't be re-ordered
- A sequentially consistent store **synchronizes-with** a sequentially consistent load of the same variable that reads the value stored
  - Does not apply to atomic operations with relaxed memory orderings
  - Advice: use sequentially consistent operations on all threads
- Noticeable performance penalty on a weakly-ordered machine with many cores
  - The overall sequence of operations must be kept consistent between the cores, possibly requiring extensive (and expensive!) synchronization operations between the processors

Non-sequentially consistent orderings
- **No single global order of events**
  - Different threads can see different views of the same operations
  - **No** mental model of operations from different threads neatly interleaved one after the other
  - Throw out mental models based on the idea of the compiler or processor reordering the instructions
- The only requirement is that **all threads agree on the modification order of each individual variable**

Understanding relaxed ordering

Call her and ask her:
- To give you a value (load) -> she will give
  - First time: any value on the list
  - Subsequent times: same value or a value from farther down the list
- To write down a value (store) -> she will always write at the bottom of the list
  - If you ask for a value later, she must give you the value you added to the list or a value from farther down the list

Sequenced before
- Within the same thread, evaluation A may be sequenced-before evaluation B
  - aka program order, but not exactly
- If A is sequenced before B (or, equivalently, B is sequenced after A), then evaluation of A will be complete before evaluation of B begins

Synchronizes-with relationship
- Appears between atomic load and store operations
- A **suitably-tagged** atomic write operation (release), W, in thread A on a variable, x, synchronizes with a **suitably-tagged** atomic read operation (acquire), R, in thread Bon x
- R reads the value stored by
  - either that write, W, or
  - a subsequent atomic write operation on x by the same thread that performed the initial write, W, or
  - a sequence of atomic read-modify-write operations on x (such as fetch_add() or compare_exchange_weak()) by any thread (where the value read by the first thread in the sequence is the value written by W)

Inter-thread happens-before
- Between threads, evaluation A inter-thread happens before evaluation B if any of the following is true

1) A **synchronizes-with** B
2) ~~A is dependency-ordered before B~~
3) A **synchronizes-with** some evaluation X, and X is **sequenced-before** B
4) A is **sequenced-before** some evaluation X, and X **inter-thread happens-before** B
5) A **inter-thread happens-before** some evaluation X, and X **inter-thread happens-before** B

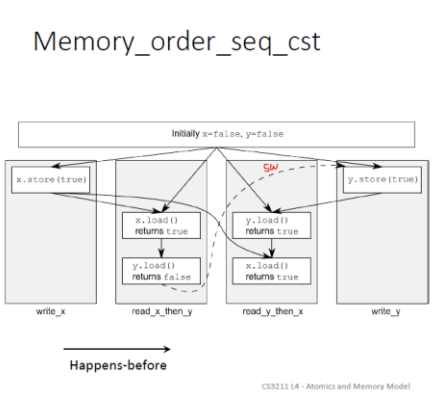
Happens-before
- Regardless of threads, evaluation A **happens-before** evaluation B if **any** of the following is true:

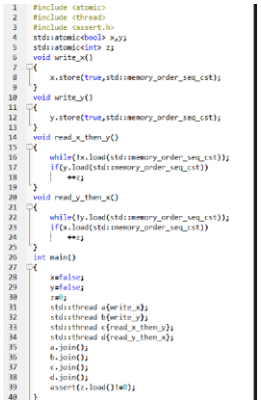
1) A is **sequenced-before** B
2) A **inter-thread happens** before B

Happens-before relationship
- Specifies which operations see the effects of which other operations
- Intra-thread happens-before
  - Based on program order, and sequenced-before
- One operation (A) occurs in a statement prior to another (B) in the source code, then A happens before B (sequenced-before)
- There is no happens-before relationship between operations that occur in the same statement
- Inter-thread happens-before
  - Based on synchronizes-with, and sequenced-before
- Transitive relationship

Visible side effects
- The side-effect A on a scalar M (a write) is visible with respect to value computation B on M (a read) if **both** of the following are true:
  - 1) A **happens-before** B
  - 2) There is no other side effect X to M where A **happens-before** X and X **happens-before** B
- If side-effect A is visible with respect to the value computation B, then the longest contiguous subset of the side-effects to M, in modification order, where B does not happen-before it is known as the visible sequence of side-effects. (the value of M, determined by B, will be the value stored by one of these side effects)
- Note: inter-thread synchronization boils down to preventing data races (by establishing happens-before relationships) and defining which side effects become visible under what conditions

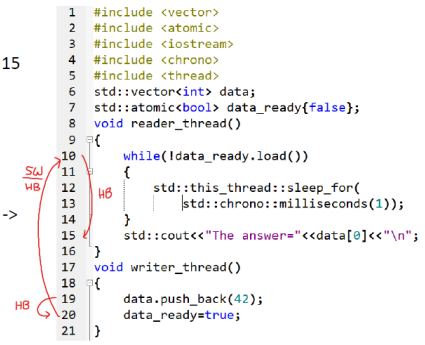
Modification order
- All modifications to any particular atomic variable occur in a total order that is specific to this one atomic variable.
- The following four requirements are guaranteed for all atomic operations:
  - 1) **Write-write coherence**: If evaluation A that modifies some atomic M (a write) happens-before evaluation B that modifies M, then A appears earlier than B in the modification order of M
  - 2) **Read-read coherence**: if a value computation A of some atomic M (a read) happens-before a value computation B on M, and if the value of A comes from a write X on M, then the value of B is either the value stored by X, or the value stored by a side effect Y on M that appears later than X in the modification order of M.
  - 3) **Read-write coherence**: if a value computation A of some atomic M (a read) happens-before an operation B on M (a write), then the value of A comes from a side-effect (a write) X that appears earlier than B in the modification order of M
  - 4) **Write-read coherence**: if a side effect (a write) X on an atomic object M happens-before a value computation (a read) B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M





Synchronizing operations and enforcing ordering

- Happens-before
  - Line 19 happens-before 20
  - Line 10 true happens-before 15
- Transitivity
- Synchronizes-with
  - Line 20 synchronizes with 10 -> 20 happens before 10



Memory ordering for atomic operations

3 memory models:
- **Sequentially consistent ordering** (memory_order_seq_cst)
- **Relaxed ordering** (memory_order_relaxed)
- **Acquire-release ordering** (memory_order_acquire, memory_order_release, memory_order_acq_rel, memory_order_consume)

Varying costs on different CPU architectures
- Additional synchro instructions are needed to achieve sequential consistent ordering over acquire-release ordering

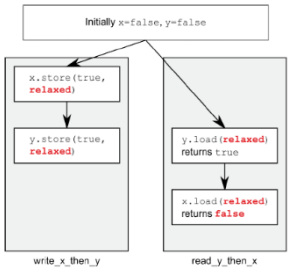
Relaxed ordering
- Operations on atomic types performed with relaxed ordering don't participate in synchronizes-with relationships
- Operations on the same variable within a single thread still obey happens-before relationships
  - Almost no requirement on ordering relative to other threads
- Accesses to a single atomic variable from the same thread can't be reordered
  - Once a given thread has seen a particular value of an atomic variable, a subsequent read by that thread can't retrieve an earlier value of the variable

Memory_order_relaxed

- Line 26: assert can fire
  - Line 14 can read false even though 13 read true
  - Even though line 8 happens before (sequenced-before) 9
- Sequenced-before relationship
  - between the stores (lines 8-9)
  - between the loads (lines 13-14)
- No happens-before between either store and either load,
  - The loads can see the stores out of order

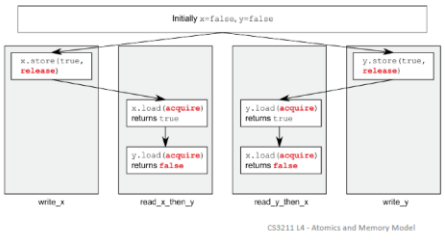
```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x_then_y()
7  {
8      x.store(true,std::memory_order_relaxed);
9      y.store(true,std::memory_order_relaxed);
10 }
11 void read_y_then_x()
12 {
13     while(!y.load(std::memory_order_relaxed));
14     if(x.load(std::memory_order_relaxed))
15         ++z;                    Even if y is true,
16 }                                we might get x = false
17 int main()
18 {
19     x=false;
20     y=false;
21     z=0;
22     std::thread a{write_x_then_y};
23     std::thread b{read_y_then_x};
24     a.join();
25     b.join();
26     assert(z.load()!=0);
27 }
```

REVIEW L3S43



Release-acquire ordering
- No total modification order, but it does introduce some synchronization
- Atomic loads use memory_order_acquire
- Atomic stores use memory_order_release
- Atomic read-modify-write operations (such as fetch_add() or exchange()) use = either acquire, release, or acquire-release(memory_order_acq_rel)
- If an **atomic store in thread A is tagged memory_order_release**, an **atomic load in thread B from the same variable is tagged memory_order_acquire**, and the load in thread B reads a value written by the store in thread A, then the **store in thread A synchronizes-with the load in thread B**.
  - All memory writes (**non-atomic and relaxed atomic**) that happened-before the atomic store from the point of view of thread A, become visible side-effects in thread B
  - That is, once the atomic load is completed, thread B is guaranteed to see everything thread A wrote to memory
  - This promise only holds if B actually returns the value that A stored, or a value from later in the release sequence
- The synchronization is established only between the threads releasing and acquiring the same atomic variable.
  - Other threads can see different order of memory accesses than either or both of the synchronized threads.

Memory_order_acquire/release

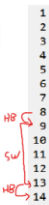



```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x()
7  {
8      x.store(true,std::memory_order_release);
9  }
10 void write_y()
11 {
12     y.store(true,std::memory_order_release);
13 }
14 void read_x_then_y()
15 {
16     while(!x.load(std::memory_order_acquire));
17     if(y.load(std::memory_order_acquire))
18         ++z;
19 }
20 void read_y_then_x()
21 {
22     while(!y.load(std::memory_order_acquire));
23     if(x.load(std::memory_order_acquire))
24         ++z;
25 }
26 int main()
27 {
28     x=false;
29     y=false;
30     z=0;
31     std::thread a(write_x);
32     std::thread b(write_y);
33     std::thread c(read_x_then_y);
34     std::thread d(read_y_then_x);
35     a.join();
36     b.join();
37     c.join();
38     d.join();
39     assert(z.load()!=0);
40 }
```

L39 assert can fire (z != 0): Both lines 17 and 23 read false as x and y are written by different threads and there is no synchronization between them.

Synchronizing with acq/rel

- Line 26: assert never fires
- Line 9 synchronizes-with 13 (true)
  - 8 happens-before 9
  - 13 happens-before 14




```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x_then_y()
7  {
8      x.store(true,std::memory_order_relaxed);
9      y.store(true,std::memory_order_release);
10 }
11 void read_y_then_x()
12 {
13     while(!y.load(std::memory_order_acquire));
14     if(x.load(std::memory_order_relaxed))
15         ++z;
16 }
17 int main()
18 {
19     x=false;
20     y=false;
21     z=0;
22     std::thread a(write_x_then_y);
23     std::thread b(read_y_then_x);
24     a.join();
25     b.join();
26     assert(z.load()!=0);
27 }
```

Since y.store(true, release) SW y.load(acquire), x must be true at L14.

Language level memory models

- Modern (C++11) and not-so-modern (Java 5) languages guarantee **sequential consistency for data-race-free programs ("SC for DRF")**
  - Compilers will insert the necessary synchronization to cope with the hardware memory model
  - SC for DRF is guaranteed only if **every** atomic only uses seq_cst
  - No guarantees for other types of orderings, even if the program is DRF
  - No guarantees if your program contains data races!

Fences

- Operations that enforce memory-ordering constraints without modifying any data and are typically combined with atomic operations that use the memory_order_relaxed ordering constraints
- Memory barriers
  - Put a line in the code that certain operations can't cross
- An **atomic_thread_fence** with **memory_order_release** ordering prevents all preceding reads and writes from moving past all subsequent stores
  - Remember: an atomic store-release operation prevents all preceding reads and writes from moving past the store-release

Atomic_thread_fence

- Line 28: assert never fires
- Line 9 synchronizes-with 15
- Line 8 happens-before 16

- Swap 8 and 9?
  - Assert can fire

```
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x_then_y()
7  {
8      x.store(true,std::memory_order_relaxed);
9      std::atomic_thread_fence(std::memory_order_release);
10     y.store(true,std::memory_order_relaxed);
11 }
12 void read_y_then_x()
13 {
14     while(!y.load(std::memory_order_relaxed));
15     std::atomic_thread_fence(std::memory_order_acquire);
16     if(x.load(std::memory_order_relaxed))
17         ++z;
18 }
19 int main()
20 {
21     x=false;
22     y=false;
23     z=0;
24     std::thread a(write_x_then_y);
25     std::thread b(read_y_then_x);
26     a.join();
27     b.join();
28     assert(z.load()!=0);
29 }
```

Chapter 5 from C++ Concurrency in Action

Tutorial 2: Atomics in C++:

Synchronisation: Mutexes vs Atomics

The atomics version is faster than the mutex version. While the mutex version has many calls to pthread_mutex_lock and _unlock around the addition to counter that may cause threads to sleep and/or contend for access, the atomics version uses a single instruction for addition: lock addl $1, actr(%rip).

seq_cst is the slowest. acquire/release and relaxed are both faster. seq_cst's store is represented in x86 by the relatively expensive *xchg* instruction (which must ensure that the processor has exclusive access to shared memory). In contrast, the other models allow the compiler to implement the store with a simple mov instruction. mov in x86 has acquire/release semantics, so it sufficient for C++'s acquire/release (but slightly overkill for C++'s relaxed).

The takeaway: choosing the right memory model may result in far greater performance!

Visible side effects

A good way to think about memory ordering is in terms of visible side effects.

```
int main() {
  int x = 5;       // A
  int y = x + 1;   // B
  return y;
}
```

```
mov     eax, 6
ret
```

In our original program, the line (A) causes a side effect (sets x to 5) that's visible to (B), because (A) precedes (B) in program order, and these two lines of code run in the same thread. Note that we can also describe this case as A is sequenced-before B. However, this side effect is not necessarily implemented as an actual store to memory! However, the program still returns the exact same value **as-if** the writes did happen. In the C++ specification, the memory model rules ultimately culminate in a set of rules detailing exactly which lines of code are visible side effects of which other lines of code under which conditions.

Threads agree on modification order

While forcing sequential consistency causes a more restrictive ordering, this does not mean that more relaxed memory models result in complete chaos. Let's look at the concept of a variable's modification order — the total order of writes to a single atomic variable.

The core principle is: **for all memory models (including relaxed consistency), all threads must agree on the same modification order for each atomic variable**. Imagine a scenario where two threads are writing to an atomic variable a, and two threads are reading from it. The threads reading from a should both observe a consistent modification order for a.

Values of 'a' seen by thread 3: 1 2 4 6 8 8 8 8 8 8
Values of 'a' seen by thread 4: 1 3 5 6 8 8 8 8 8 8

Thread 3 sees 2 -> 4 -> 6 -> 8
Thread 4 sees 3 -> 5 -> 6 -> 8

These observations are still consistent as they do not contradict each other: we do not see any write orders that are reversed between the two threads. The "real" / total modification order could have been 2 -> 4 -> 3 -> 5 -> 6 -> 8, where Thread 3 observed the 2 -> 4 and was taken out of the ready state, and Thread 4 ran and observed 3 -> 5 afterwards. **The takeaway**: we can count on a consistent modification order for each atomic variable for all threads. Note though that relative modification orders across different atomic variables may not be the same at all.

What are all the possible values printed by the code snippet?
Will your answer change if x is not atomic? (i.e., x is changed to be only an int)

4.1 Code snippet A

```
Thread 1                                  Thread 2
x.store(1, stdmo::relaxed);  // (a)       while (y.load(stdmo::acquire) != 2) { }  // (x)
y.store(2, stdmo::release);  // (b)       cout << x.load(stdmo::relaxed);          // (y)
```

The cout will print 1, and never 0.
Even if x is not atomic, the answer is the same as question 1. The same argument holds: for e.g., x = 1 cannot be reordered after the release-ordered y.store, even though x = 1 is not atomic.

4.2 Code snippet B

```
Thread 1                                  Thread 2
x.store(1, stdmo::relaxed);  // (a)       while (y.load(stdmo::acquire) != 2) { }  // (p)
y.store(2, stdmo::release);  // (b)       cout << x.load(stdmo::relaxed);          // (q)
x.store(3, stdmo::relaxed);  // (c)       while (z.load(stdmo::acquire) != 4) { }  // (r)
z.store(4, stdmo::release);  // (d)       cout << x.load(stdmo::relaxed);          // (s)
x.store(5, stdmo::relaxed);  // (e)
```

The first cout might print 1, 3, or 5. The second cout might print 3, or 5. They will never print 0.
If x is not atomic, then there are data races. So theoretically speaking, anything can happen (printing garbage, segfault, etc.). Data race between x=3 and x=5 with the first cout.

4.3 Code snippet C

```
Thread 1                                  Thread 2
x.store(1, stdmo::relaxed);  // (a)       x.store(3, stdmo::relaxed);  // (p)
y.store(2, stdmo::release);  // (b)       z.store(4, stdmo::release);  // (q)
                                          x.store(5, stdmo::relaxed);  // (r)

Thread 3
while (y.load(stdmo::acquire) != 2) { }  // (w)
cout << x.load(stdmo::relaxed);          // (x)
while (z.load(stdmo::acquire) != 4) { }  // (y)
cout << x.load(stdmo::relaxed);          // (z)
```

The first and second cout might print 1, 3, or 5. They will never print 0.
If x is not atomic, then there are data races. So theoretically speaking, anything can happen. the races are between x = 3 and x = 5 with the first cout, between x = 5 with the second cout, and finally between x = 1 with x = 3 and x = 5.

L4: Testing and Debugging Multithreaded Programs:

- Testing and debugging concurrent code is hard.
- Any type of bug is possible in concurrent code
- Focus on concurrency-related bugs
  - Unwanted blocking –easier to discover
  - Race conditions –sometimes difficult to discover

Unwanted blocking: Deadlock, Livelock, Blocking on I/O or other external input (Thread is blocked waiting for external input).

Race conditions

- **Data races**
  - Observable: undefined behavior due to unsynchronized concurrent access to a shared memory location
- **Broken invariants**
  - Dangling pointers -another thread deleted the data being accessed
  - Random memory corruption -a thread reading inconsistent values resulting from partial updates
  - Double-free -two threads pop the same value from a queue, and so both delete some associated data
- **Lifetime issues**
  - The thread outlives the data that it accesses
  - Accessing data that has been deleted or otherwise destroyed
  - Potentially the storage is even reused for another object
  - Ensure that the call to join() can't be skipped if an exception is thrown
  - Example: thread references local variables that go out of scope before the thread function has completed

Questions to use during reviewing code

- Which data needs to be protected from concurrent access?
- How do you ensure that the data is protected?
- Where in the code could other threads be at this time?
- Which mutexes does this thread hold?
- Which mutexes might other threads hold?
- Are there any ordering requirements between the operations done in this thread and those done in another? How are those requirements enforced?
- Is the data loaded by this thread still valid? Could it have been modified by other threads?
- If you assume that another thread could be modifying the data, what would that mean and how could you ensure that this never happens?

Testing multithreaded code

- Maybe the hardest type of testing
- Precise scheduling of threads is indeterminate and may vary from run to run
  - The code does not always fail (Heisenbug)
  - Difficult to reproduce problems

Guidelines for testing

- Run the smallest amount of code that could potentially demonstrate a problem: Easier to locate the faulty code if the test fails
- Eliminate the concurrency from the test to verify that the problem is concurrency-related: For bugs found "in the wild"
- Run on a multicore and single-core your multithreaded code
- Can set affinity to run single-core on multicore machine

Test various scenarios

For a queue:

- One thread calling push() or pop() on its own to verify that the queue works at a basic level
- One thread calling push() on an empty queue while another thread calls pop()
- Multiple threads calling push() on an empty queue
- Multiple threads calling push() on a full queue
- Multiple threads calling pop() on an empty queue
- Multiple threads calling pop() on a full queue
- Multiple threads calling pop() on a partially full queue with insufficient items for all threads
- Multiple threads calling push() while one thread calls pop() on an empty queue
- Multiple threads calling push() while one thread calls pop() on a full queue
- Multiple threads calling push() while multiple threads call pop() on an empty queue
- Multiple threads calling push() while multiple threads call pop() on a full queue

Test environments

- What you mean by "multiple threads" in each case (3, 4, 1024)?
- Whether there are enough processing cores in the system for each thread to run on its own core?
- Which processor architectures the tests should be run on?
- How you ensure suitable scheduling for the "while" parts of your tests?

Design for testability (in general)

- The responsibilities of each function and class are clear
- The functions are short and to the point
- Your tests can take complete control of the environment surrounding the code being tested
- The code that performs the particular operation being tested is close together rather than spread throughout the system
- You thought about how to test the code before you wrote it

Design for testability (concurrent code)

- Eliminate the concurrency -break down the code into
  - parts that operate on the communicated data within a single thread, and
  - parts responsible for the communication paths between threads
  - ensure that only one thread at a time is accessing a particular block of data
  - multiple blocks of read shared data/transform data/update shared data - testing the reading and updating of the shared data
- Watch out for library calls that use internal variables to store state
  - Use an alternate function safe for concurrent access from multiple threads

Techniques for multithreaded testing

- Stress testing (brute-force testing)
- Use special implementation of the library for synchronization primitives

Stress testing

- Run the code many times –with many threads running at once
- Write tests that maximize the problematic circumstances
- Platform to run the tests
  - Use multicore hardware
  - If available, multiple architectures, Example: on x86, load is the same independent of the memory ordering used
- Cover code paths

Special implementation of synchronization primitives library

- Mark your shared data in some way, and allow the library to check that operations on a particular shared data are done with a particular mutex locked
- Record the sequence of locks if more than one mutex is held by a particular thread at once
- Give priorities to threads to acquire a resource

Structuring multithreaded code: Try to provide suitable scheduling for your threads to execute a particular part of the code.

Testing the performance

- **Scalability** – you want your code get the expected speedup when running with increasing number of threads on increasing number of cores
- **Contention** when accessing shared data from increasing number of threads

Debugging Tools

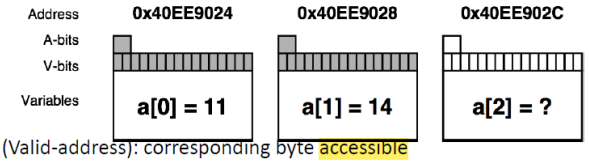
Help in identifying (concurrency) bugs

- Valgrind
  - Heavy-weight binary instrumentation: ~20x overhead
  - Designed to shadow all program values: registers and memory
  - Shadowing requires serializing threads
  - Thread error detectors: Helgrind, DRD
  - Dynamic instrumentations – don't need to compile with valgrind, just use executable
- Sanitizers
  - Compilation-based approach
- These tools need to keep track of the state of the memory
  - Compute **happens-before** to find **data races**

Valgrind memcheck –How to?

Shadow memory

- Used to track and store information on the memory that is used by a program during its execution.
- Used to detect and report incorrect accesses of memory



A-bit (Valid-address): corresponding byte accessible
V-bit (Valid-value): corresponding bit initialized

→ check if all V-bits are 1 to catch uninitialized variables

Valgrind memcheck

- Validates memory operations in a program
  - Each allocation is freed only once
  - Each access is to a currently allocated space
  - All reads are to locations already written
  - 10 –20x overhead

valgrind --tool=memcheck <prog …>

```
==29991== HEAP SUMMARY:
==29991==     in use at exit: 2,694,466,576 bytes in 2,596 blocks
==29991==   total heap usage: 16,106 allocs, 13,510 frees, 3,001,172,305 bytes allocated
==29991==
==29991== LEAK SUMMARY:
==29991==    definitely lost: 112 bytes in 1 blocks
==29991==    indirectly lost: 0 bytes in 0 blocks
==29991==      possibly lost: 7,340,200 bytes in 7 blocks
==29991==    still reachable: 2,687,126,264 bytes in 2,588 blocks
==29991==         suppressed: 0 bytes in 0 blocks
```

Helgrind in Valgrind

- Dynamic instrumentation
  - Intercepts calls to functions and instruments them
  - 100x slowdowns

Detects:

- Misuses of the POSIX pthreads API (can do C++ threads too)
  - By intercepting calls to many POSIX pthreads functions
- Potential **deadlocks** arising from lock ordering problems.
- **Data races** -accessing memory without adequate locking or synchronization.

Helgrind: Deadlock detection

- Helgrind builds a **directed graph** indicating the order in which locks have been acquired
- When a thread acquires a new lock
  - the graph is updated, and
  - then checked to see if it now contains a cycle.
- The presence of a cycle indicates a potential deadlock involving the locks in the cycle
- For 2 or multiple locks

Helgrind: Data races

- Check the order in which memory accesses can happen
  - happens-before relationship
  - builds a directed acyclic graph representing the collective happens-before dependencies
  - Monitors all memory accesses.
- No race in the case where both accesses are reads.

L4S26

Sanitizers

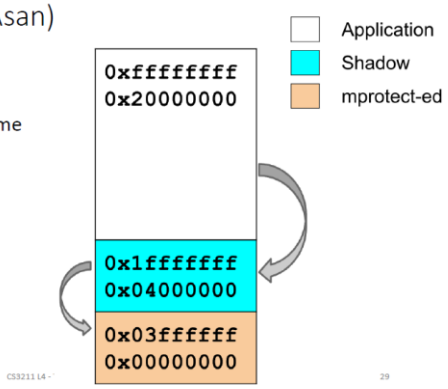
- **Compilation-based** approach to detect issues
  - clang and gcc support
  - ~5-10x overhead
  - Add "-fsanitize=address", or "-fsanitize=thread", etc. for different sanitizers

Examples:

- **AddressSanitizer** (detects addressability issues) and **LeakSanitizer** (detects memory leaks)
- **Thread sanitizers**(TSan)
- **MemorySanitizer** (detects use of uninitialized memory)
- HWASAN, or Hardware-assisted AddressSanitizer, a newer variant of AddressSanitizer that consumes much less memory
- UBSan, or UndefinedBehaviorSanitizer

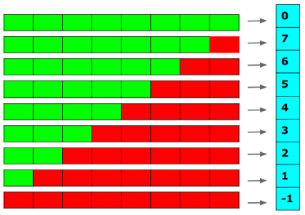
Address Sanitizer (Asan)

- 2x slowdown
- 1.5-3x memory overhead
- Hundreds of bugs in Chrome
  - Used for tests and fuzzing



Asan – how it works?

- Any aligned 8 bytes may have 9 states:
  N good bytes and 8-N bad (0<=N<=8)
- N byte access




```
*a = ...
```

```
char *shadow = addr >> 3;
if (*shadow &&
    *shadow <= ((a&7)+N-1))
  ReportError(a);
*a = ...
```

L4S30

Thread Sanitizer (Tsan)

- Instrument a running program
- Engineered for speedup –5-15x slowdown and about 5-10x memory overhead
- Discovered many races (hundreds) in Google server-side apps
  - Scales to huge apps
- Runtime library
  - Malloc replacement
  - Fully parallel
  - No expensive atomics/locks

Usage for Thread Sanitizer (TSan)

Compile and link with -fsanitize=thread, -fPIE, -pie

- Add –O2 for reasonable performance
- Run the executable (using options)
  - TSAN_OPTIONS="history_size=7 force_seq_cst_atomics=1" ./myprogram

Tsan prints a report:

WARNING: ThreadSanitizer: thread leak (pid=9509)

Thread T1 (tid=0, finished) created at:

#0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001fb33)

#1 main thread_leak3.c:10 (exe+0x000000003c7e)

Compiler-instrumentation with TSan

Instrumentation

- Function entry/exit
- Memory access

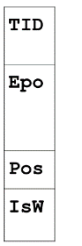
```
void foo(int *p) {
  *p = 42;
}
```

```
void foo(int *p) {
  __tsan_func_entry(__builtin_return_addre
  __tsan_write4(p);
  *p = 42;
  __tsan_func_exit()
}
```

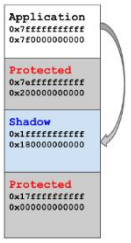
An 8-byte shadow cell represents one memory access:

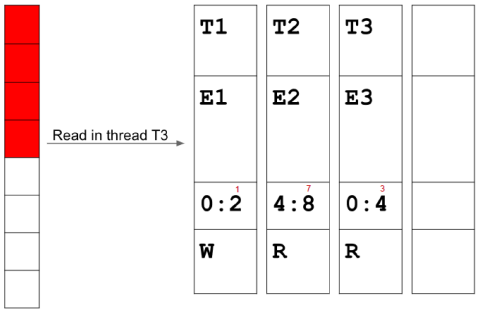
- ~16 bits: TID (thread ID)
- ~42 bits: Epoch (scalar clock)
- 5 bits: position/size in 8-byte word
- 1 bit: IsWrite

Full information (no more dereferences)



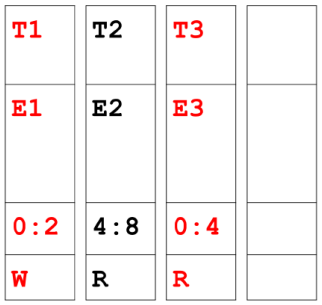

```
Shadow = 4 * (Addr & kMask);
```





- Race if E1 does not happen-before E3
- Constant time operation
  - Get TID and Epoch
  - 1 load from thread-local storage
  - 1 comparison

  (Details in FastTrack PLDI'09)



L4S41

Stack trace for previous access

- Shown in the report from Tsan
- Per-thread cyclic buffer of events
  - 64 bits per event (type + PC)
  - Events: memory access, function entry/exit
  - Information will be lost after some time
  - Buffer size is configurable
- Replay the event buffer on report
  - Unlimited number of frames

Tsan detects

- Normal data races
- Races on C++ object vptr
- Use after free races
- Races on mutexes
- Races on file descriptors
- Races on pthread_barrier_t
- Destruction of a locked mutex
- Leaked threads
- Signal-unsafe malloc/free calls in signal handlers
- Signal handler spoils errno
- Potential deadlocks (lock order inversions)

Action plan for debugging

1. Run Valgrind memcheck (and fix the errors)
2. Run Tsan (and fix the errors)
3. Run Asan (and fix the errors)
4. Run Helgrind (and fix the errors)

Summary

Modern C++ is not easy, but has many concepts that are useful for us, as developers. Concurrency in C++ is even more challenging: Built-in threads, Primitives for synchronization, Atomic<> Weapons, Testing & debugging. Avoided C++ complicated syntax, as much as possible.

Chapter 11 from C++ Concurrency in Action

Tutorial 3: Debugging Concurrent C++ Programs

Even if you don't manually manage memory with new and delete (you shouldn't), you may depend on an API that returns owned resources. For example, the Unix API often requires that you close(resource) after using them. C++ provides std::unique_ptr to protect such resources automatically using RAII.

std::unique_ptr

```
int main()
{
    // int* foo = new int { 0 };
    std::unique_ptr<int> foo = std::make_unique<int>(0);

    // FILE* bar = fopen("file.txt", "w");
    auto deleter = [](FILE* f) {
        fclose(f);
    };
    auto bar = std::unique_ptr<FILE, decltype(deleter)> {
        fopen("file.txt", "w"), deleter
    };

    use_foo(foo.get());
    use_bar(bar.get());
} // bar and foo deleted here
```

Shared Resource Challenges

- No way to track who is (not) using the resource, thus no way of knowing when to free the resource
- Each function may believe it is responsible for deleting the resource, since it is hard to know whether int* is an owned or non owning pointer.

```
void reader(int* foo)
{
    std::cout << *foo;
    delete foo;
}

void writer(int* foo)
{
    (*foo)++;
    delete foo;
}

void schedule_unsafe()
{
    int* foo = new int;

    std::thread { reader, foo }.detach();
    std::thread { writer, foo }.detach();
}
```

- **Data races**
- **Double free** - both the reader and writer delete foo without knowing if the other has done so.
- **Use after free** - the writer/reader may delete foo before reader/writer accesses it.
- **Uninitialised variable** - new int does not initialise foo with any value.

AddressSanitizer

AddressSanitizer (ASan) is a compiler and runtime technology that exposes many hard-to-find bugs with **zero** false positives.

clang++ -std=c++20 -g **-fsanitize=address** main.cpp

- Use of uninitialised memory can be detected with -fsanitize=memory.
- If you would like to see all errors, compile with clang++ -std=c++20 -g -fsanitize=address -fsanitize-recover=address -o exe main.cpp and run with ASAN_OPTIONS=halt_on_error=0 ./exe.
- compile with only 1 sanitizer at a time!: Running an executable compiled with a sanitizer with valgrind has weird effects too

Protecting resources with shared lifetimes

std::shared_ptr: is a reference-counted smart pointer that allows safely sharing resources.

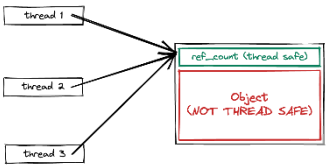
Reference counting

The idea is that each instance of the shared pointer is a reference to the pointed-to data. When you copy a shared pointer, the number of references increases by 1. When an instance is destructed, the number of references decreases by 1. Thus, when the number of references reaches 0, the std::shared_ptr automatically calls the destructor on the object that it is pointing to, essentially solving for us the problem of knowing when the last user of a resource has finished using it.

```
void rc(const char* s, const std::shared_ptr<int>& x)
{
    auto refs = x.use_count();
    printf("%s: %zu ref%s\n", s, refs, refs == 1 ? "" : "s");
}

void foo(std::shared_ptr<int> arg)  // num refs: 5
{
    rc("foo 1", arg);
    auto z = arg;   // num refs: 6
    rc("foo 2", arg);

} // num refs: 4 (`arg` & `z` are gone)

int main()
{
    auto x = std::make_shared<int>(100); // 1
    rc("initial", x);                    // 1
    {
        auto a = x;     rc("A", x); // 2
        auto b = x;     rc("B", x); // 3
        auto c = a;     rc("C", x); // 4
        foo(x);         rc("D", x); // 4
    }
    rc("E", x); // 1 (all the above are gone)
} // num refs: 0, destructor called
```

Using std::shared_ptr (BUGGY: Data race present)

```
void reader2(std::shared_ptr<int>
foo)
{ std::cout << *foo; }

void writer2(std::shared_ptr<int>
foo)
{ (*foo)++; }
```




```
void schedule_safe()
{
    std::shared_ptr<int> foo { std::make_shared<int>(0) };
    std::thread { reader2, foo }.detach();
    std::thread { writer2, foo }.detach();
}
```

Data Race: Thread safety of std::shared_ptr only applies to the reference count! int is not thread-safe. Solution: can use std::mutex, or use std::atomic.

Another buggy example 2.4.2

```
    std::shared_ptr<int> ptr;

    auto reader = std::thread([](std::shared_ptr<int>& ptr) {
        while(ptr == nullptr);
        printf("%d\n", *ptr);
    }, std::ref(ptr));

    auto writer = std::thread([](std::shared_ptr<int>& ptr) {
        for(int i = 0; i < 100; i++)
            ptr = std::make_shared<int>(i);
    }, std::ref(ptr));
```

Data race on ptr. Writer overwrites the reference to ptr, which deletes the old value and makes a new one. However, the reader may still be using the old value in *ptr for printing, even after it was freed. A better solution would be for each thread to own ptr separately by copying into the respective threads.

Another buggy example 2.4.3

```
// Doubly Linked List
struct DLLNode
{
    std::shared_ptr<DLLNode> prev;
    std::shared_ptr<DLLNode> next;
};

struct DLL
{
    std::shared_ptr<DLLNode> head {};
    std::shared_ptr<DLLNode> tail {};

    void push_front(std::shared_ptr<DLLNode>);
    void push_back(std::shared_ptr<DLLNode>);

    std::shared_ptr<DLLNode> front();
    std::shared_ptr<DLLNode> back();
};
```

Unfortunately, ThreadSanitizer cannot save us here! This is a problematic use of std::shared_ptr, even on a single thread; we get a memory leak due to a **circular reference**. Fortunately, our old friend AddressSanitizer can detect leaks as well!

For adjacent nodes in our doubly-linked list, the first node's next points at the second node, while the second node's prev points at the first – this creates a reference cycle, meaning that neither shared_ptr's reference count ever reaches 0. One solution would be to use DLLNode* prev, which does not affect the reference count of shared pointers, but keep the next pointer as a shared pointer. This way, all owning references are forward pointing and do not form a cycle.

ThreadSanitizer: unlock of an unlocked mutex

```
    ~SharedPtr2()
    {
         auto lk = std::unique_lock { *m_mutex };
         if(--(*m_count) == 0)
         {
                  delete m_ptr;
                  delete m_mutex;
                  delete m_count;
         }
    }
```

Notice that we delete m_mutex while it is locked! The guard unlocking it at the end of the destructor will thus be using m_mutex after it has been freed. In fact, ThreadSanitizer tells us about this error. To unlock the mutex safely, we should only delete it after it has been unlocked.

Shared Pointer Implementation

```
template <typename T>
struct SharedPtr3
{
private:
    size_t* m_count;
    std::mutex* m_mutex;
    T* m_ptr;

public:
    SharedPtr3(T* ptr) : m_count(new size_t(1)), m_mutex(new
std::mutex()), m_ptr(ptr)
    {
    }

    SharedPtr3(const SharedPtr3& other) : m_count(other.m_count),
m_mutex(other.m_mutex), m_ptr(other.m_ptr)
    {
         auto lk = std::unique_lock { *m_mutex };
         ++(*m_count);
    }



    ~SharedPtr3()
    {
         size_t new_count = 0;
         {
             auto lk = std::unique_lock { *m_mutex };
             new_count = --(*m_count);
         }

         /* alternatively:
                  size_t new_count = [this]() {
                     auto lk = std::unique_lock { *m_mutex };
                     return --(*m_count);
                  }();
         */

         if(new_count == 0)
             { delete m_ptr; delete m_mutex; delete m_count; }
    }

    // shouldn't need to worry about this
    T* get() { return m_ptr; }
    const T* get() const { return m_ptr; }
};
```

Notes:

- Each shared_ptr should share the same count, mutex, and ptr.
- Mutex required to synchronise the count

Atomic Version

```
struct SharedPtr4
{
private:
    std::atomic<size_t>* m_count;
    T* m_ptr;

public:
    SharedPtr4(T* ptr) :
        m_count(new std::atomic<size_t>(1)), m_ptr(ptr) {}

    SharedPtr4(const SharedPtr4& other) :
        m_count(other.m_count), m_ptr(other.m_ptr)
    {
         m_count->fetch_add(1);
    }

    ~SharedPtr4()
    {
        size_t old_count = m_count->fetch_sub(1);
        if(old_count == 1) //fetch_sub returns old value of m_count
         {
                  delete m_ptr; delete m_count;
         }
    }
};
```

Instead of using locks, we can also use atomics to achieve the same result! We simply make the reference count atomic, taking care to ensure that we share that atomic across all instances of the shared pointer. That is, we should have a pointer to an atomic, not an atomic pointer!

Extra: Overhead of std::shared_ptr

Many default to std::shared_ptr due to the flexibility of using it as a std::unique_ptr. However, this encourages several anti-patterns:

Ambiguous Ownership

Most resources should have a single owner using std::unique_ptr, borrowing references from a main thread with raw pointers. For example, you can consider rearchitecting to use Singleton pattern or Entity Component System pattern.

Memory Leaks

Premature sharing of ownership can result in unintended cyclic references, which causes memory leaks. std::unique_ptr will never have this issue, since can only ever have one owner.

Performance Overhead

Maintaining reference counts can result in unnecessary synchronisation. Similarly, initialising memory on the heap instead of on the stack can unnecessarily slow down the program and reduce cache locality.

How valgrind can help you

As compared to sanitisers, valgrind is slower but offers more details on resource usage such as memory, CPU and syscalls. First compile without -fsanitize, otherwise your program may run out of memory. Then, run the executable with valgrind ./exe to see the amount of memory leaked.

Lecture 5 –Concurrent Data Structures (in modern C++)

Design data structures for concurrency

Goal

- Multiple threads can access the data structure concurrently
  - Performing the same or distinct operations
- Each thread sees a self-consistent view of the data structure

Designing thread-safe data structures

- No data is lost or corrupted
- All invariants are upheld
- No problematic race condition

Protect the data structure with a mutex

- **Prevents true concurrent access** to the data it protects
- **Serialization**: threads take turns accessing the data protected by the mutex
- We want concurrent data structures that enable true concurrency
- Some data structures have more scope for true concurrency than others

Invariants

- Invariants –statements that are always true about a particular data structure
- Often broken during an update on the data structure
- Example: this variable no_of_itemscontains the number of items in the list
- Use invariants to reason about program correctness

Building a thread-safe data structure

- Ensure that no thread can see a state where the **invariants** of the data structure have been broken by the actions of another thread
- Take care to avoid **race conditions** inherent in the interface to the data structure by providing functions for complete operations rather than for operation steps
- Pay attention to how the data structure behaves in the presence of **exceptions** to ensure that the invariants are not broken
- Minimize the opportunities for **deadlock** when using the data structure by restricting the scope of locks and avoiding nested locks where possible

Concurrency while calling functions

- Constructors and destructors require exclusive access to the data structure
  - It's up to the users to ensure that data structures not accessed before construction is complete or after destruction has started
- Data structure that support assignment, swap(), or copy construction:
  - Decide whether these operations are safe to call concurrently with other operations or whether they require the user to ensure exclusive access even though the majority of functions for manipulating the data structure may be called from multiple threads concurrently without any problems

Truly designing for concurrency

- The smaller the protected region, the fewer operations are serialized, and the greater the potential for concurrency
- Provide the **opportunity for concurrency** to threads accessing a thread-safe data structure
  - If one thread is accessing the data structure through a particular function, which functions are safe to call from other threads?

Thread-safe data structures

- Minimize the amount of serialization that must occur
- **Lock at an appropriate granularity**
  - A lock should be held for only the minimum possible time needed to perform the required operations
- Alternatives
  - Multiple threads might perform one type of operation on the data structure concurrently, whereas another operation requires exclusive access by a single thread
  - Use std::shared_mutex to allow concurrent access from threads that read the data structure, but exclusive access for threads that modify the data structure
  - Safe for multiple threads to access a data structure concurrently if they're performing different actions, whereas multiple threads performing the same action would be problematic

Enable genuine concurrent access

- Can the scope of locks be restricted to allow some parts of an operation to be performed outside the lock?
- Can different parts of the data structure be protected with different mutexes?
- Do all operations require the same level of protection?
  - How about operations on const objects?
  - mutable keyword
  - Can a simple change to the data structure improve the opportunities for concurrency without affecting the operational semantics?

Lock-based concurrent data structures

- Ensure that the right mutex is locked when accessing the data and that the lock is held for the minimum amount of time
  - Data can't be accessed outside the protection of the mutex lock
  - There are no race conditions inherent in the interface
- Using multiple mutexes to protect separate parts of the structure
  - Deadlocks are possible

S1: A thread-safe stack with one global mutex

- Safe for multiple threads to call the member functions concurrently

```
#include <exception>
struct empty_stack: std::exception
{
    const char* what() const throw();
};
template<typename T>
class threadsafe_stack
{
private:
  std::stack<T> data;
  mutable std::mutex m;
public:
  threadsafe_stack(){}
  threadsafe_stack(const threadsafe_stack& other)
  {
    std::lock_guard<std::mutex> lock(other.m);
    data=other.data;
  }
  threadsafe_stack& operator=(const threadsafe_stack&) = delete;
  void push(T new_value)
  {
    std::lock_guard<std::mutex> lock(m);
    data.push(std::move(new_value));
  }
  std::shared_ptr<T> pop()
  {
    std::lock_guard<std::mutex> lock(m);
    if(data.empty()) throw empty_stack();
    std::shared_ptr<T> const res(
    std::make_shared<T>(std::move(data.top())));
    data.pop();
    return res;
  }
  void pop(T& value)
  {
    std::lock_guard<std::mutex> lock(m);
    if(data.empty()) throw empty_stack();
    value=std::move(data.top());
    data.pop();
  }
  bool empty() const
  {
    std::lock_guard<std::mutex> lock(m);
    return data.empty();
  }
};
```

The basic thread safety is provided by protecting each member function with a lock on the mutex, m. This ensures that only one thread is accessing the data at any one time, so provided each member function maintains the invariants, no thread can see a broken invariant.

Second, there's a potential for a race condition between empty() and either of the pop() functions, but because the code explicitly checks for the contained stack being empty while holding the lock in pop(), this race condition isn't problematic. By returning the popped data item directly as part of the call to pop(), you avoid a potential race condition that would be present with separate top() and pop() member functions such as those in std::stack<>.

Next, there are a few potential sources of exceptions. **Locking a mutex may throw an exception**, but not only is this likely to be exceedingly rare (because it indicates a problem with the mutex or a lack of system resources), it's also the first operation in each member function. Because no data has been modified, this is safe. Unlocking a mutex can't fail, so that's always safe, and the use of std::lock_guard<> ensures that the mutex is never left locked.

The call to data.push() may throw an exception if either copying/moving the data value throws an exception or not enough memory can be allocated to extend the underlying data structure. Either way, std::stack<> guarantees it will be safe, so that's not a problem either.

In the first overload of pop(), the code itself might throw an empty_stack exception, but nothing has been modified, so that's safe. The creation of res might throw an exception, though, for a couple of reasons: the call to **std::make_shared might throw** because it can't allocate memory for the new object and the internal data required for reference counting, or the copy constructor or move constructor of the data item to be returned might throw when copying/moving into the freshly allocated memory. In both cases, the C++ runtime and Standard Library ensure that there are no memory leaks and the new object (if any) is correctly destroyed. Because you still haven't modified the underlying stack, you're OK. The call to data.pop() is guaranteed not to throw, as is the return of the result, so this overload of pop() is **exception-safe**.

The second overload of pop() is similar, except this time it's the copy assignment or move assignment operator that can throw, rather than the construction of a new object and an std::shared_ptr instance. Again, you don't modify the data structure until the call to data.pop(), which is still guaranteed not to throw, so this overload is exception-safe too.

Finally, empty() doesn't modify any data, so that's exception-safe.

There are a couple of opportunities for deadlock here, because you call user code while holding a lock: the copy constructor or move constructor and copy assignment or move assignment operator on the contained data items, as well as potentially a user-defined operator new. If these functions either call member functions on the stack that the item is being inserted into or removed from or require a lock of any kind and another lock was held when the stack member function was invoked, there's the possibility of deadlock. But it's sensible to require that users of the stack be responsible for ensuring this; you can't reasonably expect to add an item onto a stack or remove it from a stack without copying it or allocating memory for it.

Because all the member functions use std::lock_guard<> to protect the data, it's safe for any number of threads to call the stack member functions. The only member functions that aren't safe are the constructors and destructors, but this isn't a problem; the object can be constructed only once and destroyed only once. Calling member functions on an incompletely constructed object or a partially destructed object is never a good idea, whether done concurrently or not. As a consequence, the user must ensure that other threads aren't able to access the stack until it's fully constructed and must ensure that all threads have ceased accessing the stack before it's destroyed.

S1: Concurrency in the thread-safe stack

- Safe for multiple threads to call the member functions concurrently
- BUT the work is serializedfor the stack data structure
  - only one thread is ever doing any work in the stack data structure at a time
- Exception safe
- Serialization limits the performance of an application that exhibits significant contention on the stack
- No means of waiting for an item to be added
  - A thread must periodically call empty(), or call pop() and catch the empty_stack exceptions
  - Consume precious resources checking for data or the user must write an external wait and notification code

Q1: A thread-safe queue with notification (one mutex)

```
template <typename T>
class threadsafe_queue
{
private:
  mutable std::mutex mut;
  std::queue<T> data_queue;
  std::condition_variable data_cond;

public:
  threadsafe_queue()
  {
  }
  void push(T new_value)
  {
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(std::move(new_value));
    data_cond.notify_one();
  }
  void wait_and_pop(T &value)
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk, [this]
                   { return !data_queue.empty(); });
    value = std::move(data_queue.front());
    data_queue.pop();
  }
  std::shared_ptr<T> wait_and_pop()
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk, [this]
                   { return !data_queue.empty(); });
    std::shared_ptr<T> res(
        std::make_shared<T>(std::move(data_queue.front())));
    data_queue.pop();
    return res;
  }
  bool try_pop(T &value)
  {
    std::lock_guard<std::mutex> lk(mut);
    if (data_queue.empty())
      return false;
    value = std::move(data_queue.front());
    data_queue.pop();
    return true;
  }
  std::shared_ptr<T> try_pop()
  {
    std::lock_guard<std::mutex> lk(mut);
    if (data_queue.empty())
      return std::shared_ptr<T>();
    std::shared_ptr<T> res(
        std::make_shared<T>(std::move(data_queue.front())));
    data_queue.pop();
    return res;
  }
  bool empty() const
  {
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
  }
};
```

The analysis for the stack applies here as well

- The wait_and_pop() functions are a solution to the problem of waiting for a queue entry
- **Exception safety**
  - If more than one thread is waiting when an entry is pushed onto the queue (data_cond.wait), only one thread will be woken by the call to data_cond.notify_one()
  - But if that thread then throws an exception in wait_and_pop() (when the new std::shared_ptr<>is constructed), none of the other threads will be woken
- Solutions:
  - Replaced with **data_cond.notify_all(),** which will wake all the threads but at the cost of most of them then going back to sleep when they find that the queue is empty after all, or
  - wait_and_pop() **call notify_one() if an exception is thrown**, or
  - **Move the std::shared_ptr<>initialization to the push() call** and storestd::shared_ptr<> instances rather than direct data values

Q2: Use shared-pointers

```
template <typename T>
class threadsafe_queue
{
private:
  mutable std::mutex mut;
  std::queue<std::shared_ptr<T>> data_queue;
  std::condition_variable data_cond;

public:
  threadsafe_queue() { }
  void wait_and_pop(T &value)
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk, [this]{ return !data_queue.empty(); });
    value = std::move(*data_queue.front());
    data_queue.pop();
  }
  bool try_pop(T &value)
  {
    std::lock_guard<std::mutex> lk(mut);
    if (data_queue.empty())
      return false;
    value = std::move(*data_queue.front());
    data_queue.pop();
    return true;
  }
  std::shared_ptr<T> wait_and_pop()
  {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk, [this]{ return !data_queue.empty(); });
    std::shared_ptr<T> res = data_queue.front();
    data_queue.pop();
    return res;
  }
  std::shared_ptr<T> try_pop()
  {
    std::lock_guard<std::mutex> lk(mut);
    if (data_queue.empty())
      return std::shared_ptr<T>();
    std::shared_ptr<T> res = data_queue.front();
    data_queue.pop();
    return res;
  }
  void push(T new_value)
  {
    std::shared_ptr<T> data(
        std::make_shared<T>(std::move(new_value)));
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
  bool empty() const
  {
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
  }
};
```

- Exception safe
- The **allocation of the new instance can now be done outside the lock in push()**
  - Beneficial for the performance of the queue
- Usage of one mutex limits the concurrency supported by this queue
  - By using the standard container (std::queue<>) you now have one data item that's either protected or not
  - For fine-grained locking you need to take control of the detailed implementation of the data structure

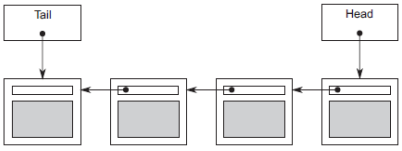
A thread-safe queue using fine-grained locks

- Analyze the constituent parts and **associate one mutex with each distinct item**
  - Insert mutexes into the data structure itself, and thus we cannot simply make use of the STL library anymore

Q3: Single-threaded queue implementation

```
template <typename T>
class queue
{
private:
  struct node
  {
    T data;
    std::unique_ptr<node> next;
    node(T data_) : data(std::move(data_))
    {
    }
  };
  std::unique_ptr<node> head;
  node *tail;

public:
  queue() : tail(nullptr)
  {
  }
  queue(const queue &other) = delete;
  queue &operator=(const queue &other) = delete;

  std::shared_ptr<T> try_pop()
  {
    if (!head)
    { return std::shared_ptr<T>(); }
    std::shared_ptr<T> const res(
        std::make_shared<T>(std::move(head->data)));
    std::unique_ptr<node> const old_head = std::move(head);
    head = std::move(old_head->next);
    if (!head)
      tail = nullptr;
    return res;
  }
```

```
  void push(T new_value)
  {
    std::unique_ptr<node> p(new node(std::move(new_value)));
    node *const new_tail = p.get();
    if (tail)
    {
      tail->next = std::move(p);
    }
    else
    {
      head = std::move(p);
    }
    tail = new_tail;
  }
};
```



- Uses **std::unique_ptr<node>** to manage the nodes
  - Ensures that they (and the data they refer to) get deleted when they're no longer needed, without having to write an explicit delete
- Problems when adding a mutex for front and back
  - push() can modify both front and back
  - push() and pop() access the next pointer of a node. If there's a single item in the queue:
    - push()updatesback->next, andtry_pop()readsfront->next
    - front==back, so bothfront->nextandback->nextare the same object → requires protection
    - You can't tell if it's the same object without reading both front and back, you now have to lock the same mutex in both push() and try_pop() → Same serialization as before

Q4:Enabling concurrency by separating data

- Pre-allocate a dummy node with no data
  - Ensure that there's always at least one node in the queue to separate the node being accessed at the front from that being accessed at the back
- Empty queue: front and back point to the dummy node
- No race on front->next and back->next
- Downside: add an extra level of indirection to store the data by pointer in order to allow the dummy nodes

```
template <typename T>
class queue
{
private:
  struct node
  {
    std::shared_ptr<T> data;
    std::unique_ptr<node> next;
  };
  std::unique_ptr<node> head;
  node *tail;

public:
  queue() : head(new node), tail(head.get())
  {
  }
  queue(const queue &other) = delete;
  queue &operator=(const queue &other) = delete;

  std::shared_ptr<T> try_pop()
  {
    if (head.get() == tail) //head is never NULL
    {
      return std::shared_ptr<T>();
    }
    std::shared_ptr<T> const res(head->data);
    std::unique_ptr<node> old_head = std::move(head);
    head = std::move(old_head->next);
    return res;
  }

  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); //dummy node
    tail->data = new_data;
    node *const new_tail = p.get();
    tail->next = std::move(p);
    tail = new_tail;
  }
};
```

- push() now access only tail, not head, which is an improvement
- try_pop() accesses both head and tail, but tail is needed only for the initial comparison, so the lock is short-lived
- the dummy node means try_pop() and push() are never operating on the same node, so you no longer need an overarching mutex

Q5: Adding locks (fine-grained thread-safe queue)

```
template <typename T>
class threadsafe_queue
{
private:
  struct node
  {
    std::shared_ptr<T> data;
    std::unique_ptr<node> next;
  };
  std::mutex head_mutex;
  std::unique_ptr<node> head;
  std::mutex tail_mutex;
  node *tail;
  node *get_tail()
  {
    std::lock_guard<std::mutex> tail_lock(tail_mutex);
    return tail;
  }
  std::unique_ptr<node> pop_head()
  {
    std::lock_guard<std::mutex> head_lock(head_mutex);
    if (head.get() == get_tail())
    {
      return nullptr;
    }
    std::unique_ptr<node> old_head = std::move(head);
    head = std::move(old_head->next);
    return old_head;
  }

public:
  threadsafe_queue() : head(new node), tail(head.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_head = pop_head();
    return old_head ? old_head->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_tail = p.get();
    std::lock_guard<std::mutex> tail_lock(tail_mutex);
    tail->data = new_data;
    tail->next = std::move(p);
    tail = new_tail;
  }
};
```

Queue invariants

- back->next==nullptr
- back->data==nullptr
- front==back implies an empty list
- A single element list has front->next==back
- For each node x in the list, where x!=back, x->data points to an instance of T and x->next points to the next node in the list. x->next==back implies x is the last node in the list
- Following the next nodes from front will eventually yield back

Q6: An even more fine-grainted lock-based queue

```
class FineQueue
{
  struct Node
  {
    std::mutex mut;
    Node *next = nullptr;
    Data data{};
  };

  std::mutex mut_back;
  Node *back;

  std::mutex mut_front;
  Node *front;

public:
  FineQueue() : mut_back{}, back{new Node()}, mut_front{},
front{back} {}

  ~FineQueue()
  {
    while (front != nullptr)
    {
      Node *next = front->next;
      delete front;
      front = next;
    }
  }
```

```
  void enqueue(Data data)
  {
    Node *new_node = new Node{};
    std::scoped_lock lock{mut_back};
    std::scoped_lock lock{back->mut};
    back->data = data;
    back->next = new_node;
    back = new_node;
  }

  std::optional<Data> try_dequeue()
  {
    Node *old_node;
    {
      std::scoped_lock lock{mut_front};
      old_node = front;
      std::scoped_lock lock{old_node->mut};
      if (old_node->next == nullptr)
      {
        return std::nullopt;
      }
      front = front->next;
    }
    Data data = old_node->data;
    delete old_node;
    return data;
  }
}
```

Line 3: Per-node mutex
- **Synchronizes-with relationship between push and pop threads**

Line 38: Additional scope introduced in try_pop()
- Avoid UAF: node_lock unlocks early, BEFORE we proceed to call delete old_node
- Delete is expensive, release mut_front early

Lock-based data structures
- Mutexes are powerful mechanisms for ensuring that multiple threads can safely access a data structure without encountering race conditions or broken invariants
- The granularity of locking can affect the potential for true concurrency

Lock-free concurrent data structures
Blocking data structures
- Algorithms and data structures that use mutexes, condition variables, and futures to synchronize the data are called **blocking** data structures and algorithms
- The application calls library functions that will suspend the execution of a thread until another thread performs an action
- These library calls are termed **blocking calls**
  - The thread can't progress past this point until the block is removed
  - Typically, the OS will suspend a blocked thread completely (and allocate its time slices to another thread) until it's unblocked by the appropriate action of another thread, such as
    - unlocking a mutex,
    - notifying a condition variable, or
    - making a future ready

Nonblocking data structures
- Data structures and algorithms that do not use blocking library functions are said to be **nonblocking**
- Types of nonblocking data structures
- Example: a spin lock is nonblocking, as it spins until the test_and_se tis successful
- **Obstruction-free**: if all other threads are paused, then any given thread will complete its operation in a bounded number of steps.
- **Lock-free**: if multiple threads are operating on a data structure, then after a bounded number of steps one of them will complete its operation.
- **Wait-free**: every thread operating on a data structure will complete its operation in a bounded number of steps, even if other threads are also operating on the data structure.

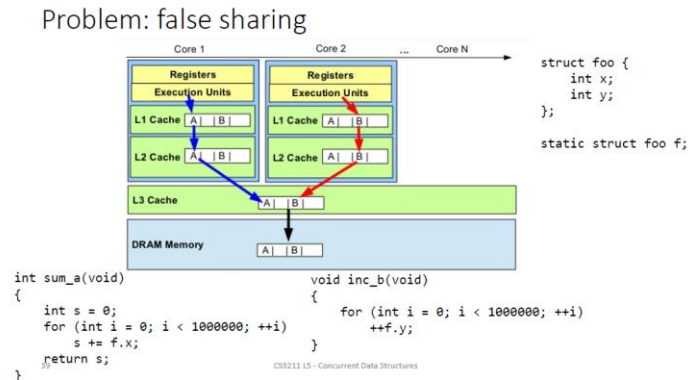
Lock-free data structures
- More than one thread must be able to access the data structure concurrently
  - They do not have to be able to do the same operations
  - If one of the threads accessing the data structure is suspended, the other threads must still be able to complete their operations without waiting for the suspended thread
  - Example: a lock-free queue might allow one thread to push and one to pop but break if two threads try to push new items at the same time

Wait-free data structures
- Data structures that avoid the following problem are wait-free
  - Lock-free algorithms with loops (using compare/exchange operations) can result in one thread being subject to **starvation**
  - If another thread performs operations with the "wrong" timing, the other thread might make progress but the first thread continually has to retry its operation
- Algorithms that can involve an unbounded number of retries because of clashes with other threads are not wait-free
- Writing wait-free data structures correctly is extremely hard
  - It's all too easy to end up writing what's essentially a spin lock

Pros of lock-free data structures
- Enable maximum concurrency
  - Some thread makes progress with every step
- Robustness
  - If a thread dies partway through an operation on a lock-free data structure, nothing is lost except that thread's data; other threads can proceed normally
  - You can not exclude threads from accessing the data structure
    - Ensure that the invariants are upheld or choose alternative invariants that can be upheld
    - Pay attention to the ordering constraints you impose on the operations
    - Ensure that changes become visible to other threads in the correct order

Cons of lock-free data structures
- Livelocks are possible
  - Two threads each try to change the data structure, but for each thread, the changes made by the other require the operation to be restarted, so both threads loop and try again
- Decrease overall performance, even though they reduce the time an individual thread spends waiting
  - Atomic operations used for lock-free code can be much slower than non-atomic operations
  - The hardware must synchronize data between threads that access the same atomic variables
    - Memory contention and write propagation
    - Cache ping-pong with multiple threads accessing the same atomic variables

Contention and cache ping pong
- If one of the threads modifies the data, this change then has to propagate to the cache on the other core, which takes time
- Depending on the memory orderings used for the operations, this modification may cause the second core to stop and wait for the change to propagate through the memory hardware
  - Extremely slow
  - Memory contention increases with the increase in number of threads
- Accessing data from the same cache line within multiple threads
  - Example: a mutex used by many threads
  - False-sharing can produce cache ping-pong as well, and it is more difficult to identify.



L5S38

Guidelines for writing lock-free code
- Use std::memory_order_seq_cst for prototyping
- Use a lock-free memory reclamation scheme
  - Use some method to keep track of how many threads are accessing a particular object and delete each object when it is no longer referenced from anywhere
  - Recycle nodes
- Watch out for the ABA problem
  - Include an ABA counter alongside the variable
  - Prevalent when using free lists
- Identify busy-wait loops and help the other thread

Summary
- Safely and efficiently using synchronization primitives and atomics to implement thread-safe data structures
- Lock-based
  - Granularity level in synchronization
- Lock-free
  - Difficult to get right
- Principles for design for concurrent data structures

Tutorial 4: Lock free programming in C++

Producers

Producers' enqueue implementation:

A. Create a new Node (new_dummy).
B. Get current work_node by reading m_queue_back
C. Set the new job in the work_node.
D. Point work_node at new_dummy by setting next pointer. This also simultaneously converts the work node to a "real" job-holding node.
E. Also update m_queue_back so other producers know where the new end of the queue is.

- Need to perform three writes to publicly accessible memory in steps C,D,E. If we make this lock free, even if we have no "data races" (by using atomic etc), we may still have race conditions.
- Race condition happens when two producers attempt to place their job in the same dummy node, thus overwriting the other producer's job.

This creates at least two potential problems:

1. A producer may get in another producer's way, eg. by overwriting their job.
2. A producer may not be correctly synchronised with consumers, causing them to read an invalid state.

We can solve problem (1) by choosing (E) to be the "definitive" source of truth for which producer is allowed to modify which nodes.

```
// queue field:
std::atomic<Node*> m_queue_back;

// doesn't work:
{
    Node* new_dummy = new Node();
    Node* work_node = m_queue_back;
    m_queue_back = new_dummy;
    // ... modify work_node
}

// works:
{
    Node* new_dummy = new Node();
    Node* work_node = m_queue_back.exchange(new_dummy);
    // ... modify work_node
}
```

In other words, we combine (B) and (E) into one atomic step. In doing so, each producer will now get their own Node to work with, and we avoid them stepping on each other's toes.

To solve problem (2), we need to release our writes (the job) to the consumer in a way that they can reliably acquire. To do this, we will use release-acquire semantics that we have discussed previously. We will use a node's next pointer as the shared memory location. If we perform a release-write to work_node->next, any consumers that read that node's ->next will synchronise-with the producer. If they read a nullptr, then there isn't a valid job (it's a dummy).

Note that the invariant of this data structure is that the last node (m_queue_back) always points to a dummy node. All non-dummy nodes will have a non-null next pointer, and only the dummy node has a null next pointer.

This results in the following algorithm:

- **[A]** Create a new Node (new_dummy). This will be the queue's new dummy node after we are done.
- **[B+E]** Simultaneously (atomically) get the current dummy node, and also update the queue to point to our new dummy node: m_queue_back.exchange(new_dummy)
- **[C]** Set our job the node we got from the above step (work_node).
- **[D]** Point our work_node at our new_dummy by release-storing its next pointer. This also simultaneously converts work_node to a "real" job-holding node, and also serves to synchronise-with the corresponding consumer that reads this pointer.

Consumers

Since we are writing a lock-free queue, we will not implement a blocking dequeue, only a non-blocking try_pop.

At a high level, these are the steps that we need to do:

A. Read m_queue_front for the front of the queue.
B. Check whether the node is a dummy or not, by reading its next pointer.
   i. If it's a dummy, then we're at the end of the queue, so we return an empty result.
   j. Otherwise, continue – there are some jobs in the queue.
C. Update m_queue_front to point at the next node (ie. m_queue_front = m_queue_front->next)
D. Return the job in the node we just removed from the queue (ie. return old_front->job)

Just like the case of producers, there are at least two potential race conditions, even if all the publicly accessible memory is atomic:

1. A consumer may get in another consumer's way, eg. by consuming a node meant for another consumer
2. A consumer may not be correctly synchronised with producers, causing them to read an invalid state.

From the previous section, we know that we can solve problem (2) by synchronising the producer with us before we read the job, so we know we have to perform step (B) with an acquire memory ordering.

What about problem (1)? We solved it previously by ensuring no two producers operate on the same node by using an atomic exchange to perform the read and update operations at the same time. Unfortunately, we cannot do that so easily here:

```
// The following cannot be made atomic
Node* node_to_consume = m_queue_front.exchange(
  m_queue_front->next
  // There is an entirely separate atomic load here
  // ... furthermore we only want to exchange if it's not null
  // so this code doesn't even do what we need it to do
);
```

The main problem is that we need to only perform the exchange conditionally, and we need to load the new value we want to exchange from the pointer we are trying to exchange itself…

The Compare-And-Swap Pattern

In order to solve this problem, we must use the compare-and-swap pattern, sometimes known as the compare-exchange pattern. This is another **atomic** operation, similar to the existing ones we've seen so far (load, store, fetch_add, etc.).

It first performs a comparison on the memory location, checking if the current value is the same as our expected value. If it isn't, then we give up. If it is, then we store our new value. In either case, we get the old/current value back. All of this is done in one atomic instruction.

The best way to reason about compare-and-swap is that it presents the operation of "set X from OLD to NEW", rather than simply "set X to NEW". This statement doesn't make sense if X was not OLD before setting, and so the compare-and-swap should fail. In this way, we are performing the exchange only if nobody else has done anything to it in the meantime — ensuring that our idea of reality and the actual reality match before we actually write anything.

Implementing fetch_add with the CAS Loop Pattern

The true power of compare-and-swap becomes apparent when we put it in a loop. Indeed, most algorithms that use compare-and-swap typically use them in a loop, since we (usually) have to handle the failure case (ie. when the current value is not our expected value).

This is how we might implement atomic fetch_add using compare-and-swap:

```
int fetch_add(std::atomic<int> &value, int add)
{
  // first, load the old value.
  int old_value = value.load();
  while (true)
  {
    // this is the new value which we want to store.
    int new_value = old_value + 5;

    // set value *FROM old_value* TO new_value.
    // if it succeeded (returns true), we're done.
    if (value.compare_exchange_weak(old_value, new_value))
      return old_value;

    // here, we failed -- someone else changed `value` so that
    // it was no longer `old_value`.

    // `old_value` is taken by reference, so we get the current
    // value back even on failure (and so we don't need to load it
ourselves).
    // go back to the top of the loop and retry, with our refreshed
`old_value`.
  }
}
```

As mentioned in the comments, one thing to keep in mind is that compare_exchange takes in the old_value as a non-const reference. Whether or not the CAS succeeded, we still would have had to load value (to do the compare), and so we conveniently get that old value returned back to us.

obj→compare_exchange_weak(*expected, desired)

Atomically compares the object representation value representation of the object pointed to by obj with that of the object pointed to by expected, and if those are bitwise-equal, replaces the former with desired (performs read-modify-write operation). Otherwise, loads the actual value pointed to by obj into *expected (performs load operation).

Strong vs Weak compare-and-swap

The weak version can spuriously fail. That is, it can fail to perform the exchange even though the current value matched our expected value. The strong version is not susceptible to such weaknesses. In general, the guidance is that the _strong version can be more expensive. If you are already using CAS in a loop and each iteration of the loop is relatively cheap, then you should probably use the _weak version. In our case, the only update we had to do was a simple + 5, we used _weak. On the other hand, if updating the expected state on failure requires heavy computation, or the CAS is not even being used in a loop, then you should prefer the _strong version.

Of course, we would prefer if compare_exchange couldn't fail spuriously, at all. One of the main reasons that a weak version exists is due to hardware architectures.

On x86, we have the cmpxchg instruction, which is guaranteed by the architecture to not fail spuriously. However, other architectures use a different primitive, often called load-linked/store-conditional (LL/SC), to implement compare-and-swap.

The load of the current value is the "load-linked"; the CPU automatically handles "breaking" the LL/SC relationship (making the conditional store fail) if anybody updates the load-linked location, thus giving us the compare-and-swap behaviour. Unfortunately, some CPU architectures will invalidate the LL/SC even if nobody stored to the location. Common cases include:

A context switch
Another load-linked instruction
Another store instruction

Implementing try_pop with a CAS loop

The key idea behind our implementation is that if any other thread managed to take a job while we were trying to, then we fail and try again — instead of leaving the queue in a broken state.

This gives us the following high-level algorithm:

Read old_front = m_queue_front for the front of the queue.
Loop:
Check whether the node is a dummy or not by reading the next pointer.
If it's a dummy, then we're at the end of the queue, so we return nullopt.
Otherwise, continue
Update m_queue_front to point at the next node with CAS:
m_queue_front.compare_exchange_weak(old_front, next)
If successful, break out of the loop
Return the job in the node we just removed from the queue: return old_front->job

```
struct Job
{
  int id;
  int data;
};

// (Incorrect)
// Unbounded lock free queue with push and non-blocking try_pop
// This design is incorrect as it suffers from both ABA and UAF.

class JobQueue1
{
  using stdmo = std::memory_order;

  // A node is a dummy node if its next pointer is set to QUEUE_END
  // We use the next ptr to establish the synchronizes-with relationship
  // next is in charge of "releasing" job
  struct Node
  {
    std::atomic<Node *> next = QUEUE_END;
    Job job;
  };

  static inline Node *const QUEUE_END = nullptr;

  // Avoid false sharing with alignment
  alignas(64) std::atomic<Node *> m_queue_back;  // producer end
  alignas(64) std::atomic<Node *> m_queue_front; // consumer end

public:
  // Queue starts with a dummy node
  JobQueue1() : m_queue_back(new Node()),
m_queue_front(m_queue_back.load(stdmo::relaxed))
  {
  }

  ~JobQueue1()
  {
    // Assumption: no other threads are accessing the job queue
    Node *cur_node = m_queue_front.load(stdmo::relaxed);
    while (cur_node != QUEUE_END)
    {
      Node *next = cur_node->next;
      delete cur_node;

      cur_node = next;
    }
  }

public:
  void push(Job job)
  {
    Node *new_dummy = new Node();

// Use m_queue_back.exchange to establish a global order of all  enqueues
    // We use acq_rel because:
    // - Release initialisation of `new_dummy`
    // - Similarly acquire initialisation of `work_node`
    // initialisation = what the Node constructor does
    Node *work_node = m_queue_back.exchange(new_dummy, stdmo::acq_rel);

    // now, work_node is unique for every producer (push)
    work_node->job = job;    // First write the job

    // "release" job to consumers, and also append to LL at the same time
    work_node->next.store(new_dummy, stdmo::release);
  }

  std::optional<Job> try_pop()
  {
    // Splice node from the front of queue, but only if it's not dummy
    // Successfully splicing a node establishes global order of pops

    Node *old_front = m_queue_front.load(stdmo::relaxed);
    while (true)
    {
      // "Acquire" job if it exists
      // Also use next pointer to know what to update m_queue_front to
      Node *new_front = old_front->next.load(stdmo::acquire);
      if (new_front == QUEUE_END)
      {
        // Observed dummy node, so we can abort as the queue is empty
        // (or close to it)
        return std::nullopt;
      }

      // for now, we use relaxed.
      if (m_queue_front.compare_exchange_weak(old_front,new_front, //
                                              stdmo::relaxed))
      {
        break; // Node now belongs to us
      }

      // We couldn't update m_queue_front, so someone else successfully
      // poped a node. We'll just loop.
    }

    Job job = old_front->job;
    delete old_front;

    return job;
  }
};
```

Problem #1: the 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.

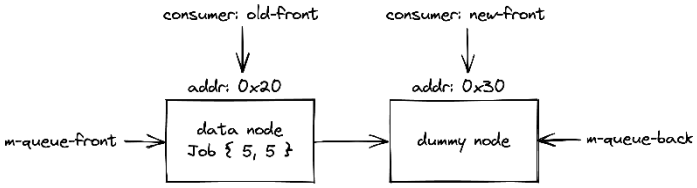
But if old_front can be set to another value, and then set back to the same value again in time for the CAS operation to be performed, the CAS would succeed! This is because previously freed memory addresses may have been allocated to subquent calls to new Node().

This is where the name comes from: a value initially has value A, is set to B, and then back to A.

This is not always a problem. For example, our fetch_add implementation above really does work, as we don't rely on the property that A actually never changed in the loop. Instead all we really care about is that the new value is 5 higher than the old value.

In our case, we DO care that the queue has not changed, since it's important that new_front really does refer to the second node in the queue. We only have this guarantee if the queue really did not change between the start and end of the loop.

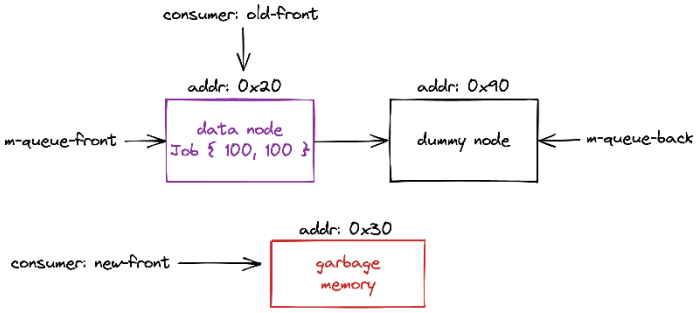
First, a consumer tries to pop a node, and sees this state before entering the CAS loop:



Now, the OS is mean to our consumer and puts it to sleep. In the meantime, the queue goes through many pushes and pops, allocating and deallocating nodes as it goes. It still holds a pointer to (what it thinks is) the old_front, address 0x20.

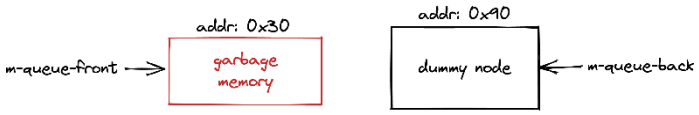
However, while it may point to a node, the identity of the node is now different! Previously we had Job{5, 5} — but now it's Job{100, 100}, a completely different node that just happened to have the same address.

On the other hand, new_front (the second node in the queue when the consumer last checked) might have already been deallocated, and now the pointer points to garbage:



Now, the consumer wakes up. It tries to perform the compare-and-swap, and succeeds. This is because the pointer value of old_front is the same as the pointer value of m_queue_front — even though the identity of the node has changed!

After succeeding at the CAS, we set the queue's front to point at what used to be the second node, but now points at garbage:



Solving ABA with generation-counted pointers

There are several ways to solve the ABA problem, but is most commonly solved by including a "**generation counter**" alongside whatever data we're performing the CAS on.

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.

We use a 64-bit integer for our counter; this means that, for us to get a false positive on the compare-and-swap (leading to the ABA problem), the following conditions must hold:

- a new m_queue_front was allocated at the same address as our old one
- 2^64 (18446744073709551616) pops have happened (causing the counter to overflow and wrap around) while our consumer was put to sleep

This is highly unlikely, mainly due to the second condition. So with **very high probability**, we can avoid the ABA problem by using generation counters.

```
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
```

Note that we don't need to do this for the back pointer; since we are not using a compare-and-swap loop for push, we are not vulnerable to the ABA problem.

```
GenNodePtr old_front = m_queue_front.load(stdmo::relaxed);
while(true)
{
    // this part is similar -- just need an additional `.node` get the
    // node out from the GenNodePtr
    Node* old_front_next = old_front.node->next.load(stdmo::acquire);
    if(old_front_next == QUEUE_END)
        return std::nullopt;

    // note that the generation is strictly increasing
    GenNodePtr new_front { old_front_next, old_front.gen + 1 };

    // this part is also similar, except we CAS with the GenNodePtr
       instead of just a simple Node*.
    if(m_queue_front.compare_exchange_weak(old_front, new_front,
         stdmo::relaxed))
    {
        break; // Node now belongs to us
    }
}
```

The most important thing here is that the generation of our new_front (if we succeeded at the CAS) is "newer" than the old front.

Extra: lock-free-ness of GenNodePtr

How can we be sure that our GenNodePtr is actually lock-free? The keen-eyed among you might have already noticed that std::atomic<T> doesn't actually make any guarantees about being lock-free, only that it is atomic.

There's two ways we can tell: the **is_lock_free** method on an 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. We used a static_assert on our GenNodePtr to make sure.

16-byte atomic compare-and-swap is done with the cmpxchg16b instruction on x86_64. This isn't supported by some very old x86_64 CPUs, so note that we had to pass -march=native to the compiler to ensure that it uses this instruction; otherwise, our static assertion will fail.

Problem #2: use-after-free (UAF)

While we won't succeed at performing a CAS when other consumers have changed the queue, have changed the queue, we're still dereferencing a pointer from a possibly stale value of m_queue_front (which we stored as old_front). This can cause data races as it's possible a new node is allocated again on the same address (regardless of generation!), and now we can have the constructor of std::atomic race with a use of the same object with .load().

Solutions to this problem generally fall under a few classes:

- Never free anything (ie. just leak all the memory).
- Mark nodes for deletion while there are still threads in try_pop, and when the last one leaves, we free all of them at once.
- Use reference counting (ie. an atomic shared_ptr) to know when there are no more remaining references to a particular object.
- Use hazard pointers to track which threads have references to which objects

We'll go with a variant of solution (2). Rather than trying to delete when the last try_pop leaves, which can be very rare, we simply won't try to delete nodes at all while the queue is alive, but store them somewhere so that we can delete them all at once when the queue is being destructed.

However, this would still allow a memory leak to occur, and our memory usage can keep increasing as the queue is used more. To avoid this, we'll reuse nodes that were deleted and put them back in the queue when we need new nodes.

This essentially functions as a "free list" of nodes that we recycle for future use. We only allocate new nodes from the heap when our recycling centre has been exhausted. Note that this means that the peak usage of our queue can still be quite high, and if the queue is very long for only a short time, we'll keep those recycled nodes around doing nothing. However, this is still a relatively simple solution, so we'll stick with it.

The C++ standard library actually has a specialisation of std::atomic for shared_ptr. So, why don't we use it? Well, the problem is that it's **not lock-free**. None of the 3 major implementations of the STL (libc++, libstdc++, and MSVC's STL) have a lock-free implementation of this — it just uses mutexes under the hood. Note that nowhere in the specification does it say that std::atomic must actually be lock-free! (Only std::atomic_flag is guaranteed to be lock-free).

Implementing the Recycling Centre

Since we don't care about the order that recycled nodes are used, we can just use a stack for it, rather than a queue. However, since we still want the queue overall to be lock-free, **our recycling centre also needs to be lock-free**! First, for the recycling node stack, we just need a single pointer to keep track of the top of the stack:

```
// we use these as sentinel values.
static inline Node* const QUEUE_END = nullptr;
static inline Node* const STACK_END = QUEUE_END + 1;

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
alignas(64) std::atomic<GenNodePtr> m_recycled_stack_top; // recycled
node stack
```

We added a new sentinel value to use as the "empty stack" value. Before we forget, we still need to clean up the stack in the queue's destructor, as promised. It's relatively simple:

```
// we need to clean up the recycled nodes as well
cur_node = m_recycled_stack_top.load(stdmo::relaxed).node;
while(cur_node != STACK_END)
{
    Node* next = cur_node->next;
    delete cur_node;
    cur_node = next;
}
```

Now for the fun part, which is allocating a new node (or getting one from our recycling stack):

```
// either get a node from the recycling stack if we have some,
// or allocate a new one if we don't.
Node *get_recycled_node_or_allocate_new()
{
  GenNodePtr old_stack_top = m_recycled_stack_top.load(stdmo::relaxed);
  while (true)
  {
    // if we have no more recycled nodes, return a newly-allocated one.
    if (old_stack_top.node == STACK_END)
      return new Node();

    Node *cur_stack_next = old_stack_top.node->next.load(stdmo::relaxed);
    GenNodePtr new_stack_top{cur_stack_next, old_stack_top.gen + 1};

    if (m_recycled_stack_top.compare_exchange_weak( //
            old_stack_top,                          //
            new_stack_top,                          //
            stdmo::relaxed))
    {
      // successfully got a node from the recycling centre
      return old_stack_top.node;
    }
  }
}
```

Next, we need a way to put things into this recycling stack when we pop nodes from our queue. The implementation is very similar to our classic CAS-loop-with-generation-counter pattern:

```
// Put node in recycling centre
void add_node_to_recycling_stack(Node *node)
{
  // Standard CAS loop with generation counter to avoid ABA.
  GenNodePtr old_stack_top = m_recycled_stack_top.load(stdmo::relaxed);
  while (true)
  {
    node->next.store(old_stack_top.node, stdmo::relaxed);
    GenNodePtr new_stack_top{node, old_stack_top.gen + 1};

    if (m_recycled_stack_top.compare_exchange_weak( //
            old_stack_top,                          //
            new_stack_top,                          //
            stdmo::relaxed))
    {
      break;
    }
  }
}
```

Using the recycling centre

All that's left is to use these functions when we create and delete nodes (instead of calling new and delete directly). That's fairly straightforward:

```
void push(Job job)
{
    Node* new_dummy = get_recycled_node_or_allocate_new();
    new_dummy->next.store(QUEUE_END, stdmo::relaxed);
    // the rest is the same...
}
```

Note that for push, we have to explicitly reset the next pointer to our sentinel, because it might have been recycled and have stale data inside.

```
std::optional<Job> try_pop()
{
  // most of it is the same… only this part changes
  Job job = old_front.node->job;
  add_node_to_recycling_stack(old_front.node);

  return job;
}
```

Can we still get a use-after-free situation?

No, we can't. By definition, we never really free nodes, only add them to the recycling centre. Thus, we will never end up dereferencing a deallocated node, which was our UAF problem in the first place.

We might end up checking the ->next of a node that's already been recycled, but that's fine — it simply won't match with what we see in the queue, and we'll retry our CAS loop.

Solving UAF with recycling

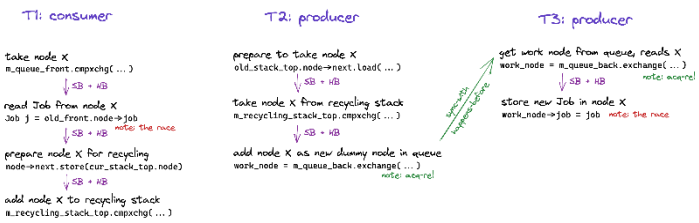
In our recycling stack implementation, we've used **memory_order_relaxed**, since the nodes in the stack don't contain any data other than the next pointer. We can imagine that threads only need to observe m_recycling_stack_top, and that atomicity gives us the guarantees we need.

Unfortunately, our environmentally-friendly recycling centre has a data race:
That corresponds to these two pieces of code:

```
work_node->job = job;          Job job = old_front.node->job;
```

Take careful note of the ordering of the race here. According to ThreadSanitizer, we tried to read the job first, and then perform the write. While it's true that when a data race occurs, it's hard to say which one happened "first", the order that ThreadSanitizer presents is useful to keep in mind as it's usually the more intuitive ordering of events.

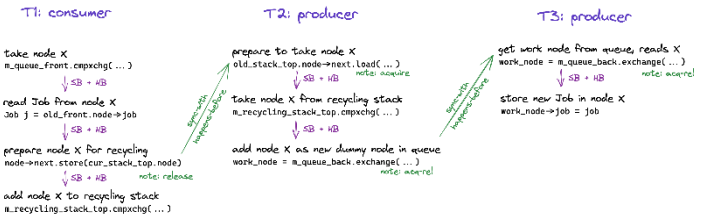
In this ordering, it would suggest that the read in try_pop can potentially see a FUTURE value, written by a subsequent push. This is now possible because of node recycling! To make this race more obvious, let's draw out our favourite synchronisation diagram:



As we can indeed see, there's no synchronisation between loading the job in T1 and storing it in T3.

Demo 4: Correct queue implementation

To resolve this, we can make T1 synchronise-with T2 during the load/store of node->next. The data race happens when the node that T1 adds to the recycling stack is the one that T2 takes out. Thus, if T2 does indeed take out node X, it would have read the value stored by T1. By using release-acquire here, we would have established a synchronises-with relationship:



With the transitive nature of happens-before, we now have a correct synchronisation between T1 and T3. To do this, we just need to change a few lines of code:

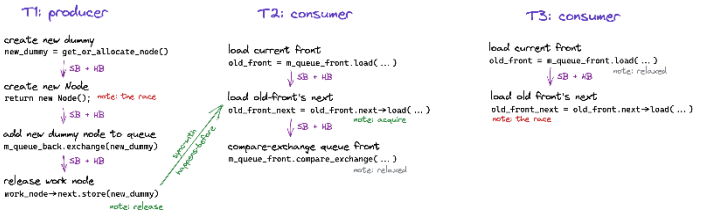
```
Node *get_recycled_node_or_allocate_new()
{
  GenNodePtr old_stack_top = m_recycled_stack_top.load(stdmo::relaxed);
  while (true)
  {
    if (old_stack_top.node == STACK_END)
    {
      return new Node();
    }

    // here: use **acquire**. synchronise with the release-store of
    // node->next in `add_node_to_recycling_stack`
    Node *cur_stack_next = old_stack_top.node->next.load(stdmo::acquire);
    // the rest is the same...
  }
}
void add_node_to_recycling_stack(Node *node)
{
  GenNodePtr old_stack_top = m_recycled_stack_top.load(stdmo::relaxed);
  while (true)
  {
    // here: use **release**. synchronise with the acquire-load of
    // node->next in `get_recycled_node_or_allocate_new`
    node->next.store(old_stack_top.node, stdmo::release);
    // the rest is the same...
  }
}
```

Problem #4: Internal data race

If we run with the modified test code below that forces the producer thread to yield after pushing every item (allowing consumers to "overtake" it): Then we actually get another race! If we look at the line numbers that ThreadSanitizer complains about, we get a race.

This is what TSan claims is the "writer":
return new Node();

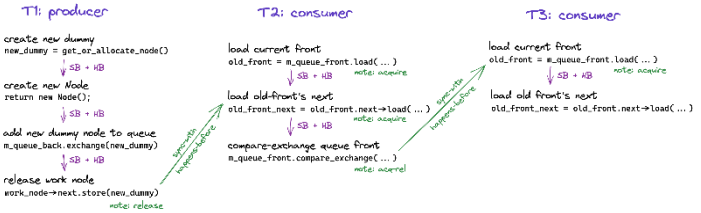
And this is what it claims is the "reader":
Node* old_front_next = old_front.node->next.load(stdmo::acquire);



There is indeed no synchronise-with relationship between T1 and T3. This means that we could potentially get a race between the reader on the right and the writer on the left. Even though this might seem like it violates causality (and/or involves time travel), it is still a data race.

Demo 5: The final fix

In order to fix this, we need to find a way to synchronise T2 and T3, so that T1 will synchronise-with T3 as well. We can do this by making the compare-exchange on the front-of-queue be acquire-release instead, and the loading of the old_front be an acquire load:



```
#include <atomic>
#include <optional>

struct Job
{
  int id;
  int data;
};

// (is correct)
// Unbounded lock free queue with push and non-blocking try_pop
// Avoids ABA using generation counter
// Avoids UAF by reusing nodes instead of deallocating them
class JobQueue5
{
  using stdmo = std::memory_order;

  // A node is a dummy node if its next pointer is set to QUEUE_END
  // We use the next ptr to establish the synchronizes-with relationship
  // next is in charge of "releasing" job
  struct Node
  {
    std::atomic<Node *> next = QUEUE_END;
    Job job;
  };

  // we use these as sentinel values.
  static inline Node *const QUEUE_END = nullptr;
  static inline Node *const STACK_END = QUEUE_END + 1;

  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
  alignas(64) std::atomic<GenNodePtr> m_recycled_stack_top; // recycled node
stack

public:
  // Queue starts with a dummy node
  JobQueue5() //
      : m_queue_back(new Node()),
m_queue_front(GenNodePtr{m_queue_back.load(stdmo::relaxed), 1}),
m_recycled_stack_top(GenNodePtr{STACK_END, 1})
  {
  }

  ~JobQueue5()
  {
    // Assumption: no other threads are accessing the job queue
    Node *cur_node = m_queue_front.load(stdmo::relaxed).node;
    while (cur_node != QUEUE_END)
    {
      Node *next = cur_node->next;
      delete cur_node;
      cur_node = next;
    }

    // we need to clean up the recycled nodes as well
    cur_node = m_recycled_stack_top.load(stdmo::relaxed).node;
    while (cur_node != STACK_END)
    {
      Node *next = cur_node->next;
      delete cur_node;
      cur_node = next;
    }
  }

  // either get a node from the recycling stack if we have some,
  // or allocate a new one if we don't.
  Node *get_recycled_node_or_allocate_new()
  {
    GenNodePtr old_stack_top = m_recycled_stack_top.load(stdmo::relaxed);
    while (true)
    {
      if (old_stack_top.node == STACK_END)
        return new Node();

      // use **acquire**. synchronise with the release-store of
      // node->next in `add_node_to_recycling_stack`
      Node *cur_stack_next = old_stack_top.node->next.load(stdmo::acquire);

      GenNodePtr new_stack_top{cur_stack_next, old_stack_top.gen + 1};

      if (m_recycled_stack_top.compare_exchange_weak( //
              old_stack_top,                          //
              new_stack_top,                          //
              stdmo::relaxed))
      {
        // successfully got a node from the recycling centre
        return old_stack_top.node;
      }
    }
  }

  // Put node in recycling centre
  void add_node_to_recycling_stack(Node *node)
  {
    GenNodePtr old_stack_top = m_recycled_stack_top.load(stdmo::relaxed);
    while (true)
    {
      // use release. synchronise with the acquire-load of
      // node->next in `get_recycled_node_or_allocate_new`
      node->next.store(old_stack_top.node, stdmo::release);

      GenNodePtr new_stack_top{node, old_stack_top.gen + 1};

      if (m_recycled_stack_top.compare_exchange_weak( //
              old_stack_top,                          //
              new_stack_top,                          //
              stdmo::relaxed))
      {
        break;
      }
    }
  }

public:
  void push(Job job)
  {
    Node *new_dummy = get_recycled_node_or_allocate_new();
    new_dummy->next.store(QUEUE_END, stdmo::release);

    Node *work_node = m_queue_back.exchange(new_dummy, stdmo::acq_rel);

    work_node->job = job;
    work_node->next.store(new_dummy, stdmo::release);
  }

  std::optional<Job> try_pop()
  {
    // most of it is the same...
    // but we changed this load to acquire
    GenNodePtr old_front = m_queue_front.load(stdmo::acquire);
    while (true)
    {
      Node *old_front_next = old_front.node->next.load(stdmo::acquire);
      if (old_front_next == QUEUE_END)
        return std::nullopt;

      GenNodePtr new_front{old_front_next, old_front.gen + 1};

      // and we change this CAS to be acquire-release
      if (m_queue_front.compare_exchange_weak(old_front, //
                                              new_front, stdmo::acq_rel))
      {
        break;
      }
    }

    Job job = old_front.node->job;
    add_node_to_recycling_stack(old_front.node);
    return job;
  }
};
```

Queue Benchmarks

Now that we have seen both a fine-grained-lock queue and a lock-free queue, how else can we learn which queue a situation calls for, if not through benchmarks!

The specific form of benchmarking we will be looking at is **microbenchmarking**; microbenchmarks track the performance of a single well-defined task, and is most useful for CPU work that is run many times (also known as hot code paths).

In this section, we will be benchmarking 3 concurrent queue implementations:

- a basic coarse-grained-lock queue
- the fine-grained-lock queue from Lecture 5
- the lock-free queue we just wrote

Benchmark setup and metrics

When benchmarking a piece of code, we first decide the metric by which we evaluate our code. Some common examples include CPU cycles, MFLOPS, and real time (wall-clock-time taken from the start to the end of a program).

To understand how each queue scales under contention with a different number of producers and consumers, we will measuring the real time taken for a few different setups for each queue:

- Single Producer, Single Consumer (SPSC)
- Single Producer, Multiple Consumers (SPMC)
- Multiple Producers, Single Consumer (MPSC)
- Multiple Producers, Multiple Consumers (MPMC)

In this setup, we use a **std::barrier** to ensure all producers and consumers threads have reached the start of the benchmark loop before starting the timer and commencing execution. When all threads have finished executing, we end the timer. Within the loop, we busily spin on a variable in between enqueues / dequeues to simulate work done on the threads in an effort to emulate real world usage.

Try running the code below on Fsmbolt (or locally if possible!) and observe how each queue performs in different setups. Note that the execution time is about 30 seconds so please be patient.

Running with simulated work between consecutive enqueue/dequeues
SPSC benchmarks:
SPSC CoarseLockedJobQueue took: 260130µs
SPSC FineLockedJobQueue took: 233000µs
SPSC LockFreeJobQueue took: 266310µs

SPMC benchmarks:
1P2C CoarseLockedJobQueue took: 251312µs
1P2C FineLockedJobQueue took: 367206µs
1P2C LockFreeJobQueue took: 296053µs

1P4C CoarseLockedJobQueue took: 726801µs
1P4C FineLockedJobQueue took: 365763µs
1P4C LockFreeJobQueue took: 335175µs

MPSC benchmarks:
2P1C CoarseLockedJobQueue took: 324418µs
2P1C FineLockedJobQueue took: 958496µs
2P1C LockFreeJobQueue took: 429071µs

4P1C CoarseLockedJobQueue took: 867296µs
4P1C FineLockedJobQueue took: 2915416µs
4P1C LockFreeJobQueue took: 771916µs

MPMC benchmarks:
2P2C CoarseLockedJobQueue took: 484619µs
2P2C FineLockedJobQueue took: 698458µs
2P2C LockFreeJobQueue took: 604591µs

4P4C CoarseLockedJobQueue took: 1577709µs
4P4C FineLockedJobQueue took: 1466190µs
4P4C LockFreeJobQueue took: 1175500µs
8P8C CoarseLockedJobQueue took: 3701999µs
8P8C FineLockedJobQueue took: 3390677µs
8P8C LockFreeJobQueue took: 3067856µs

Lecture 6 - Concurrency in Go

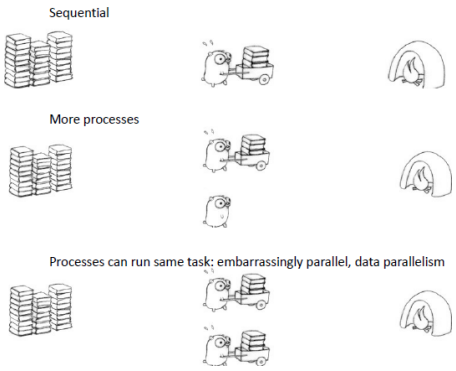
Until now in CS3211 (C++):	Next in CS3211 (Go):
Model your program in terms of *threads*	Model your program in terms of tasks
Synchronize the access to the *memory* between them	Synchronize the tasks by making them communicate
Use *thread pools* to limit the number of threads that must be handled by the machine	

Go
- Programming language announced at Google in 2009
- Compiled programming language
- Statically typed
- (Partially) syntactically similar to C, but with
  - Memory safety
  - Garbage collection
  - **CSP-style concurrency**
- Compilers & tools: gc, gccgo, gollvm

Concurrent designs
- Types of parallelism
- Limiting factors for parallelism

Solutions to a problem

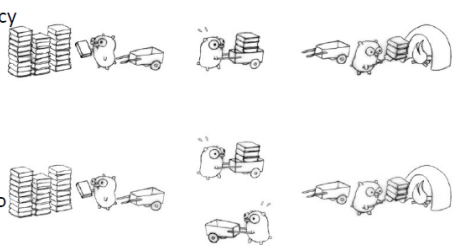
1. With only one gopher this will take too long
2. More gophers are not enough
   - need more carts
3. More gophers and carts
   - bottlenecks at the pile and incinerator
   - need to synchronize the gophers.



Concurrent composition
- Not automatically parallel!
  - However, it's automatically parallelizable!
- This can be twice as fast when running in parallel

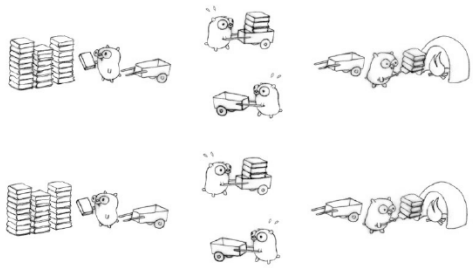
Concurrent designs

- Three gophers in action, but with (likely) delays
- Finer-grained concurrency
- Four distinct gopher procedures:
  - load books onto cart
  - move cart to incinerator
  - unload cart into incinerator
  - return empty cart
- Enables different ways to parallelize



Concurrency enabled more parallelization

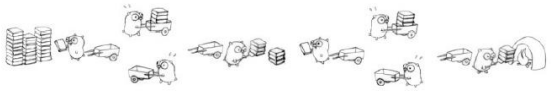
- Now parallelize on the other axis
  - 8 gophers



Other concurrent designs

- Design 1

- Design 2



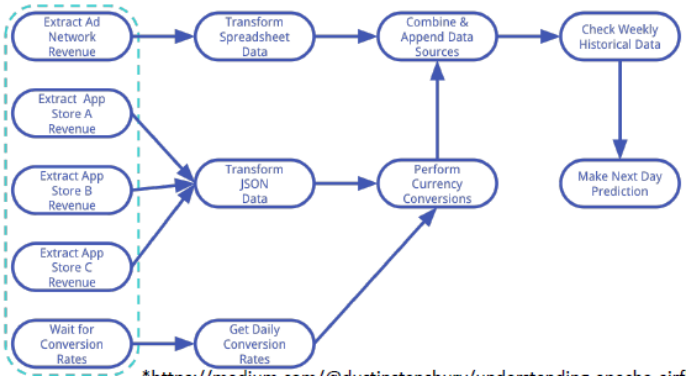
Back to computing
- In our book transport problem, substitute:
  - book pile => web content
  - gopher => CPU
  - cart => rendering, or networking
  - incinerator => proxy, browser, or other consumer
- It becomes a concurrent design for a scalable web service
  - Gophers are serving web content

Take-away points
- There are many concurrent designs
  - Many ways to break the processing down
- **Finer level of granularity enables our program to scale dynamically when it runs to the amount of parallelism possible on the program's host**
  - Amdahl's law in action!

Types of parallelism
- **Task parallelism**
  - Do the same work faster
- **Data parallelism**
  - Embarrassingly parallel algorithms
  - Do more work in the same amount of time

Task Dependency Graph
- Can be used to visualize and evaluate the task decomposition strategy
- A **directed acyclic graph**:
  - Node: Represent each task, node value is the expected execution time
  - Edge: Represent control dependency between task

Properties:
- Critical path length: maximum (slowest) completion time
- Degree of concurrency = Total Work / Critical Path Length
  - An indication of amount of work that can be done concurrently



Concurrent Programming Challenges
- Finding enough concurrency
- Granularity of tasks
- Coordination and synchronization

Parallel Program: Speedup

- Measure the benefit of parallelism
  - A comparison between sequential and parallel execution time

$$S_p(n) = \frac{T_{best\_seq}(n)}{T_p(n)}$$

Won't see ideal speedup of p duie to overhead of synchronisation, memory contention, etc.

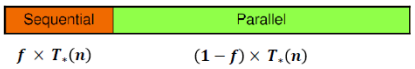
Amdahl's Law (1967)

Speedup of parallel execution is limited by the fraction of the algorithm that cannot be parallelized (f).

- f($0 \leqslant f \leqslant 1$) is called the **sequential fraction**
- Also known as **fixed-workload performance**
- The most well-known law for discussing speedup performance
  - Applicable at all levels of parallelism

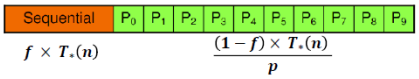
Amdahl's Law: Implication

- Sequential execution time:



$f \times T_*(n)$ $\quad (1-f) \times T_*(n)$

- Parallel execution time:



$f \times T_*(n)$ $\quad \frac{(1-f) \times T_*(n)}{p}$

$$S_p(n) = \frac{T_*(n)}{f \times T_*(n) + \frac{1-f}{p} T_*(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

Concurrency + Communication

- Go model –based on **Communicating Sequential Processes (CSP)***
  - Concurrency: structure a program by breaking it into pieces that can be executed independently
  - Communication: coordinate the independent executions

*C. A. R. Hoare: Communicating Sequential Processes (CACM 1978)

Ideas of CSP

- Refined to process calculus
- Can be used to reason about program correctness

Abstractions in Go

Concurrency

- Goroutines
  - A function running independently
  - Spin up (start) a goroutine using go function_name
- Run on OS threads
- Tasks

Communication

- Channels
  - Goroutines can write to and read from a channel (channel<-, <-channel)
- select statements
- Dependencies

Goroutines

- Function running **independently**
  - In the **same address space** as other goroutines
  - Like & in shell
- **Cheaper than threads**
- Goroutines follow the **fork-join** model (but they don't actually join)

Running Goroutines

- Runtime *multiplexes* goroutines onto OS threads
  - Automatic scheduling –mapping M:N
  - **Decouples concurrency from parallelism**
  - They run in parallel, but you have more potential for concurrency because you have several tasks. How many tasks you have is independent of how many execution threads you have
- Goroutine is a special class of coroutine (concurrent subroutine)
  - When a goroutine blocks, that thread blocks
    - but no other goroutine blocks
- Preemptable: Go's runtime can suspend them

Goroutines example

Line 3: the main goroutine is automatically created and started when the process begins

Line 4: start a goroutine using keyword go

Line 8: the print might never happen because the main goroutine finishes execution before sayHellocompletes

Line 11: anonymous function

```
3  func main() {
4      go sayHello()
5  // continue doing other things
6  }
7  func sayHello() {
8      fmt.Println("hello")
9  }
```

```
11 go func() {
12     fmt.Println("hello")
13 }()
14 // continue doing other things
```

Goroutines are lightweight

- A newly minted goroutine is given a few kilobytes, which is almost always enough
  - When it isn't, the runtime grows (and shrinks) the memory for storing the stack automatically
- The CPU overhead averages about three cheap instructions per function call
- It is practical to create hundreds of thousands of goroutines in the same address space
- Goroutines are not garbage collected! (but memory is garbage collected)
  - Programmer should prevent goroutine leaks

Goroutines

- Line 12: Join point
- Line 7: The goroutine is running a closure that has closed over the iteration variable salutation

```
3  func main() {
4      var wg sync.WaitGroup
5      for _, salutation := range []string{"hello", "greetings", "good day"} {
6          wg.Add(1)
7          go func() {
8              defer wg.Done()
9              fmt.Println(salutation)
10         }()
11     }
12     wg.Wait()
13 }
```

Go runtime is observant enough to know that a reference to the salutation variable is still being held, and therefore will **transfer the memory to the heap** so that the goroutines can continue to access it.

Line 19: write this loop is to pass a copy of salutation into the closure

```
13     var wg sync.WaitGroup
14     for _, salutation := range []string{"hello", "greetings", "good day"} {
15         wg.Add(1)
16         go func(salutation string) {
17             defer wg.Done()
18             fmt.Println(salutation)
19         }(salutation)
20     }
21     wg.Wait()
22 }
```

Potential issues with shared memory

- Need to synchronize access to shared memory locations
  - Similar to what we did in C++
  - **sync** package
- **Don't rely on shared memory** for memory locations that are modified
  - Never modify a shared memory location
- Use **channels** instead of modifying shared memory

Channels

- Serves as a conduit for a stream of information: Like the pipe (|) in shell
- Values may be passed along the channel, and then read out downstream
  - Pass a value into a chan variable, and then somewhere else in your program read it off the channel
- No knowledge is required about the other parts of your program that work with the channel
- A **channel is a reference to a place in memory where the channel resides**
  - Channels (references of channels) can be passed around your program
- Typed
- Bi-/uni-directional
- Blocking
  - Write to a channel that is full waits until the channel has been emptied
  - Read from a channel that is empty waits until at least one item is placed on it
  - Can cause **deadlocks**!

Creating a channel

- Lines 4-5: declare and create a bidirectional channel using built-in make function

```
4  var dataStream chan interface{}
5  dataStream = make(chan interface{})
```

- Lines 8-9: declare a unidirectional channel

```
8  var receiveChan <-chan interface{}
9  var sendChan chan<- interface{}
10 dataStream := make(chan interface{})
11 // Valid statements:
12 receiveChan = dataStream
13 sendChan = dataStream
```

- Line 22: receive will block until timerChan delivers.
  - Value sent is other goroutine's completion *time*

```
16 timerChan := make(chan time.Time)
17 go func() {
18     time.Sleep(deltaT)
19     timerChan <- time.Now()
20 }()
21 // Do something else; when ready, receive.
22 completedAt := <-timerChan
```

Blocking operations

- Lines 23, 26: blocking read and write
- Line 25: ok Boolean indicates whether the read was
  - a value generated by a **write**, or
  - a default value generated from a **closed** channel

```
21  stringStream := make(chan string)
22  go func() {
23      stringStream <- "Hello channels!"
24  }()
25  salutation, ok := <-stringStream
26  fmt.Printf("(%v): %v", ok, salutation)
```

- Line 33: reading from a closed channel
  - Allowed any number of times

```
31  intStream := make(chan int)
32  close(intStream)
33  integer, ok := <- intStream
34  fmt.Printf("(%v): %v", ok, integer)
```

When reading from a closed channel, will get default value. Line 34 prints: false 0

Synchronizing using channels

- Line 48: *ranging* over a channel
  - The loop doesn't need an exit criteria

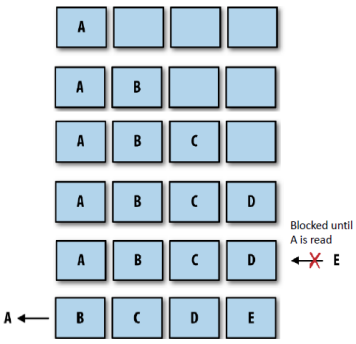
```
41      intStream := make(chan int)
42      go func() {
43          defer close(intStream)
44          for i := 1; i <= 5; i++ {
45              intStream <- i
46          }
47      }()
48      for integer := range intStream {
49          fmt.Printf("%v ", integer)
50      }
```

- Line 62: instead of writing n times to the channel to unblock each goroutine, you can simply close the channel

```
51  begin := make(chan interface{})
52  var wg sync.WaitGroup
53  for i := 0; i < 5; i++ {
54      wg.Add(1)
55      go func(i int) {
56          defer wg.Done()
57          <-begin
58          fmt.Printf("%v has begun\n", i)
59      }(i)
60  }
61  fmt.Println("Unblocking goroutines...")
62  close(begin)
63  wg.Wait()
```


Buffered channel

`c := make(chan rune, 4)`



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

Ownership of a channel

- Owner is the goroutine that instantiates, writes, and closes a channel
- Useful when reasoning about program correctness
- Unidirectional channels
  - Owners have a write-access view into the channel (chan or chan<-)
  - Utilizers only have a read-only view into the channel (<-chan)

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	

Ownership increases safety

- Because we're the one initializing the channel, we remove the risk of deadlocking by writing to a nil channel
- Because we're the one initializing the channel, we remove the risk of panicing by closing a nil channel
- Because we're the one who decides when the channel gets closed, we remove the risk of panicing by writing to a closed channel
- Because we're the one who decides when the channel gets closed, we remove the risk of panicing by closing a channel more than once
- We wield the type checker at compile time to prevent improper writes to our channel

select statement

- **Compose channels together in a program to form larger abstractions**
- Bind together channels
  - locally, within a single function or type,
  - globally, at the intersection of two or more components in a system
- Help safely bring channels together with concepts like cancellations, timeouts, waiting, and default values
- Similar in syntax with a switch block
  - BUT **case statements aren't tested sequentially**, and **execution won't automatically fall through if none of the criteria are met**

Behavior of select

- All channel reads and writes (case statements) are considered simultaneously to see if any of them are ready
  - populated or closed channels in the case of reads
  - channels that are not at capacity in the case of writes
- The **entire select statement blocks if none of the channels are ready**
- Handle the following:
  - Multiple channels have something to read
  - There are never any channels that become ready
  - We want to do something, but no channels are currently ready

Multiple channels have something to read

Output:
c1Count: 505
c2Count: 496

```
for i := 1000; i >= 0; i-- {
    select {
    case <-c1:
        c1Count++
    case <-c2:
        c2Count++
    }
}
fmt.Printf("c1Count: %d\nc2Count: %d\n", c1Count, c2Count)
```

Go runtime will perform a pseudorandom uniform selection over the set of case statements: Each case has an equal chance of being selected

Channels are not ready

- Never ready: timeout
- Line 44: time.After returns a channel that sends the current time after a time.Duration
- Do work while waiting: use default

```
41  var c <-chan int
42  select {
43  case <-c:
44  case <-time.After(1 * time.Second):
45      fmt.Println("Timed out.")
46  }
```

- block at select until something becomes available at c, or a second has passed.

For-select loop

- Allows a goroutine to make progress on work while waiting for another goroutine to report a result.
- Have some work that needs to be done while waiting for something → we can do processing while waiting.
- Once done is closed, case ← done will enter and break the loop.

```
21  done := make(chan interface{})
22  go func() {
23      time.Sleep(5*time.Second)
24      close(done)
25  }()
26  workCounter := 0
27  loop:
28  for {
29      select {
30      case <-done:
31          break loop
32      default:
33      }
34      // Simulate work
35      workCounter++
36      time.Sleep(1*time.Second)
37  }
38  fmt.Printf("%v cycles of work.\n", workCounter)
```

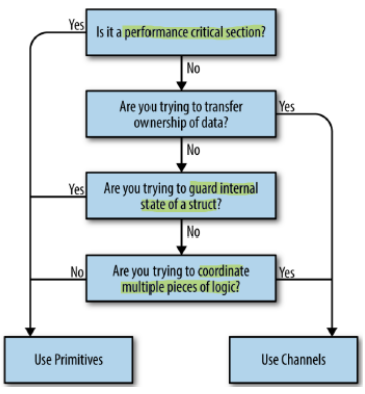
The sync package

Regarding mutexes, the sync package implements them, but we hope Go programming style will encourage people to try higher-level techniques. In particular, consider structuring your program so that only one goroutine at a time is ever responsible for a particular piece of data.

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

Sync package

- Used mostly in small scopes such as a struct
- Contains
  - **WaitGroup**: wait for a set of concurrent operations to complete
  - **Synchronization primitives**
    - Mutex and RWMutex
    - Cond
    - Once
  - **Basic constructs**
    - Pool



The Go memory model
- Specifies the conditions under which reads of a variable in one goroutine can be guaranteed to observe values produced by writes to the same variable in a different goroutine
  - Happens Before
  - Synchronization of goroutines and channels

Happens before
- **Within a single goroutine, reads and writes must behave as if they executed in the order specified by the program** (sequenced before)
- The **execution order observed by one goroutine may differ from the order perceived by another** (weak memory model)
- To guarantee that a read r of a variable v observes a particular write w to v, ensure that w is the only write r is allowed to observe. That is, r is guaranteed to observe w if **both** of the following hold:
  - w happens before r.
  - Any other write to the shared variable v either happens before w or after r.
- The **happens before relation is defined as the transitive closure of the union of the sequenced before and synchronized before relations**

Synchronized before
- The go statement that starts a new goroutine **synchronized before** the goroutine's execution begins
- The exit of a goroutine is not guaranteed to be synchronized before any event in the program
- A **send on a channel is synchronized before the completion of the corresponding receive** from that channel.
- The **closing of a channel is synchronized before a receive that returns a zero value** because the channel is closed
- A **receive from an unbuffered channel is synchronized before the send on that channel completes**
- The **kth receive on a channel with capacity C is synchronized before the (k+C)th send** from that channel completes.

Send-receive Synchronized Before

- A *send* on a channel is **synchronized before** the completion of the corresponding receive from that channel.


```
var c = make(chan int, 10)
var a string

func f() {
    A  a = "hello, world"
    B  c <- 0
}

func main() {
    C  go f()
    D  <-c
    E  print(a)
}
```

- A *receive* from an unbuffered channel is **synchronized before** the send on that channel completes


```
var c = make(chan int)
var a string

func f() {
    A  a = "hello, world"
    B  <-c      receive
}

func main() {
    C  go f()
    D  c <- 0   send
    E  print(a)
}
```

Summary
- Go helps distinguish between concurrency and parallelism
- Using a different way to implement concurrency based on CSP
  - Goroutines and channels

Tutorial 5: Goroutines and Channels

So far, we have covered synchronization **with shared memory systems**. While sharing memory is convenient and fast due to hardware support, it is easy to introduce memory safety issues such as **data races** and **use after free**.

For the remainder of the module, we explore 2 alternative approaches:
- Message passing to not share memory (Go)
- Using a borrow checker to guarantee lifetime and exclusive access at compile time (Rust)

Naïve Concurrent Counter

```
package main
import "fmt"
func main() {
    count := 0
    go func() {
        count++
    }()
    go func() {
        count++
    }()
    fmt.Println("Count: ", count)
}
```

count is likely to be 0. "(the) exit of a goroutine is not guaranteed to happen before any event in the program."

Recall that in C++, std::thread panics if not joined on exit, and std::jthread automatically joins on exit. However, goroutines do not come equipped with join handles, because language designers wanted to encourage asynchronous synchronization.

The most common way to synchronize goroutines to achieve a "joined" event is to use **sync.WaitGroup**. Similar to a barrier, sync.WaitGroup blocks a waiting goroutine until some number of tasks are completed.

A common usage of wait groups is as follows:
- **Main thread creates var wg sync.WaitGroup**
- **Main thread synchronously increments wg.Add(1)** for each unit of work to do **before spawning each goroutine**
- Main thread then spawns goroutines to complete some work. Within the goroutines, they may decrement the count by wg.Done() any number of times.
- **Main thread wg.Wait()s for the count to reach 0** after spawning worker goroutines.

```
func main() {
  count := 0
  var wg sync.WaitGroup
  for i := 0; i < 1000; i++ {
      wg.Add(1) // add BEFORE spawning
      go func() {
          defer wg.Done() // RAII. called on exit
          count++ //DATA RACE
      }()
  }
  wg.Wait() // wait until all 1000 goroutines are done

  fmt.Println("Count: ", count)
}
```

The above code has a data race! There is a data race due to:
- same memory location
- no enforced ordering between access from different threads
- one or more of those accesses is not atomic
- one or more accesses is a write

We can detect this with Go's equivalent of C++'s ThreadSanitizer: -race

Attempt #1: Exclusive Access

Similar to C++, Go has sync.Mutex and atomic.Uint64. Since we are learning about message passing, let's use chan (channel) to achieve the same result. Conceptually, shared memory maintains invariants on fully replicated state, whereas message passing maintains invariants on disjoint / partially replicated state. What does this mean?
- In shared memory, count is shared, so we must protect it by granting exclusive access with a mutex
- In message passing, count can be passed between goroutines, so we can guarantee **no data races via exclusive ownership**.

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

In this code snippet, we created an "unbuffered" channel using make(chan int). The channel made this way is blocking. When main goroutine sends 0 to channel ch, it will be blocked waiting for a consumer of the data. Any goroutine may consumes from the channel using <-ch. main can then progress up to wg.Wait(). The first goroutine consuming from ch will read the data into variable count, increment it, then block on enqueuing it back to ch. The other goroutines in the loop will then repeat this process sequentially. After enqueing the data, each goroutine calls wg.Done() upon exiting. Lastly, main goroutine read the final value off ch after all the goroutines are Done.

It turns out that there is a deadlock here.
Consider the interaction between the last worker and main goroutine
- A is blocking on D to receive from <-ch
- D cannot run until C is done Wait()ing
- C will wait until B is Done()
- B is called only after A has ch <-

This cyclic dependency forms a deadlock!

Solution 1: We can break the cyclic dependency by reordering A and B.

```
wg.Done()   // B
ch <- count // A

wg.Wait()                    // C
fmt.Println("Count: ", <-ch) // D
```

This is safe because
- C can only be unblocked by the last worker, which must have acquired the last count.
- Even if main proceeds from C after B is Done(), but before that last worker has enqueued at A, main will block at D for the last count to be enqueued.

Note that it is not safe to reorder C and D, because we want main to read count only after all worker goroutines are Done().

Solution 2: We can also use buffered chan to resolve our deadlock.
However, we **do not recommend** using buffered channels to resolve deadlocks because it simply kicks the can down the road
- When the buffered channel fills up, it blocks just like an unbuffered channel
- Permutations of interleavings between producers-producers, producer-consumer and consumer-consumer increases
- There is now more than one "owner" to values

```
func main() {
    ch := make(chan int, 1) // make buffered channel of size 1
    var wg sync.WaitGroup
```

Recall that the worker was blocking at A for main to dequeue at D. By changing make(chan int) into make(chan int, 1), the channel can now "buffer" 1 item before blocking. This results in A being a non-blocking operation, which breaks our cyclic dependency.

Attempt #2: Merging Counts
Imagine you are working on Google Docs with your teammates and you each have to acquire an exclusive ownership/access to the document before editing/viewing it. Fortunately, many applications such as our counter and Google Docs can scale linearly - by independently computing and merging into a final result.

Consider the following system design of a YouTube view counter. There are:
- Multiple producers of counts (eg viewers)
- Multiple consumers of counts (eg SEA/EU/US servers)
- Multi-producer multi-consumer message queue (eg Kafka) connecting producers and consumers
- Main thread (eg database) should reflect the total count at the end of the process (eg servers switched off for the night)

count has 2 properties we can exploit to independently compute local sums and merge into a global sum:
- **Associativity**: $(x + y) + z = x + (y + z)$, ie we can add local sums in any order and still arrive at the same global sum
- **Identity**: $0 + x = x$, ie all local sums should begin with 0

Solution 1.1
In Go, it is diomatic for the last writer to close the channel, so as to signal to readers there is no more data. However, our reader knows exactly how many writers there are, so it can read exactly N sums instead of using for-range. Afterwards, whether main closes the channel or not (becomes garbage collected), there will be
- Not more than one close() of channel
- No more writers sending on channel

```
package main
import (
    "fmt"
    "time"
)
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) {
    local_sum := 0
    for {
        select {
        case cnt := <-q:
            local_sum += cnt
        case <-done: // until stopped (channel closed)
            sumCh <- local_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
    }
    close(sumCh)
    fmt.Println("Sum: ", sum)
}
```

```
// collect all sums
sum := 0
for subSum := range sumCh {
    sum += subSum
}
fmt.Println("Sum: ", sum)
```

Note this would not work as sumCh is never closed. So main never stops reading from for-range. Note that consumer() cannot close(sumCh) right after sending because
- Multiple close() from different consumers results in a panic
- Consumer X may send sumCh <- sum after consumer Y close(sumCh), resulting in a panic

Solution 1.2
Consider our YouTube system design: In Solution 1.1, we made each view server communicate with the database by yet another MPSC queue. Instead, servers can communicate directly with the database via individual TCP streams.

Just like with count, we can avoid synchronizing close() between N writers by giving each writer an exclusive sumCh to send local sums back to main. Then, each writer can safely close() their sumCh[i] exactly once after they are done sending.

```
func producer(done chan struct{}, q chan int) {
    for {
        select {
        case q <- 1:
        case <-done:
            return
        }
    }
}
func consumer(done chan struct{}, q chan int, sumCh chan<- int) {
    sum := 0
    for {
        select {
        case <-done:
            sumCh <- sum
            close(sumCh)
            return
        case num := <-q:
            sum += num
        }
    }
}
var (
    NumProducer = 5
    NumConsumer = 5
)

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

    sumChs := make([]chan int, 0, NumConsumer) // 1 channel per consumer
    for i := 0; i < NumConsumer; i++ {
        sumCh := make(chan int, 1)
        sumChs = append(sumChs, sumCh)
    }

    for i := 0; i < NumProducer; i++ {
        go func() {
            <-start
            producer(done, q)
        }()
    }
    for j := 0; j < NumConsumer; j++ {
        j := j // capture j in the scope
        go func() {
            <-start
            consumer(done, q, sumChs[j])
        }()
    }

    close(start)            // signal to all goroutines to start
    time.Sleep(time.Second) // run for 1 second
    close(done)             // signal to all goroutines they should exit

    // collect all sums
    sum := 0
    for _, ch := range sumChs {
    // NOTE: range over slice, not combined channel
        sum += <-ch
    }
    fmt.Println("Sum: ", sum)
}
```

2 Revisiting Multi-Producer Multi-Consumer Queue
In the last tutorial, we took a look at how to write a lock-free MPMC queue. Fortunately, Go provides a thread-safe MPMC queue chan as a primitive.

Unfortunately, a queue made of chan is not lock-free: in chan's implementation, you will find mutex locking at both the sender and the receiver.

By the definition of lock-free in the textbook: *If multiple threads are operating on a data structure, then after a bounded number of steps one of them will complete its operation.*

This does not hold: when there is no receiver(sender), all senders(receivers) will be blocked.

Blocking Queue
For the producer, we simulate viewers that are sequentially sending counts of 1 to q.

NOTE that our implementation may block producers (ie users/viewers) on a full q <-. In reality, we want a truly unbounded queue or backpressure mechanism.

Recap:
- for { } is Go's equivalent to while true
- select is Go's blocking equivalent of switch. It polls on some numbers of channels and "switch"es to any non-blocking ones.
- When a channel is close()d, receiving on it immediately returns buffered values or zero value (and hence is selectable). You can manually check if a channel is closed with **val, isClosed := <- ch**.
- NOTE that if there are multiple cases available for selection, any 1 of the case will be selected. You **cannot rely on your code's order**. This was implemented to prevent starving any 1 case.

Goroutines are a resource too, and we should not leak it after using the MPMC queue. We can signal to producers and consumers to end by close()ing a shared done channel from main.

```
for i := 0; i < nProducer; i++ {
    go producer(done, q)
}
time.Sleep(time.Second) // runs for 1 second
close(done)             // stop all producers and consumers
```

Non-Blocking Queue
If we want to make the queue non-blocking, we can use the select-default idiom. We make a queue type out of chan int. We then implement try "class" methods for non-blocking enqueue and dequeue operations.

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

func (q *queue) tryDequeue() (int, bool) {
    select {
    case num, ok := <-q:
        return num, ok
    default:
    }
    return 0, false
}
```

The default case in select provides an exit for the goroutine when all the cases are blocked, effectively making accessing the channel asynchronously.

2.3 Extra: Use of context.Context
When querying asynchronous resources (eg DB, HTTP request), you often want a goroutine that synchronously waits for the result, yet is cancellable manually, via a timeout or upon a deadline. Go 1.7 introduces the **context** package to **select** all these options into a single Done() channel.

```
func main() {
    // cancel (2nd return value) can be used to manually cancel() before
    // the timeout
    ctx, _ := context.WithTimeout(context.Background(), time.Second)

    q := make(chan int)
    sumChs := make([]chan int, 0, NumConsumer)
    for i := 0; i < NumConsumer; i++ {
        sumCh := make(chan int)
        sumChs = append(sumChs, sumCh)
    }

    for i := 0; i < NumProducer; i++ {
        go func() {
            producer(ctx, q)
        }()
    }
    for j := 0; j < NumConsumer; j++ {
        j := j
        go func() {
            consumer(ctx, q, sumChs[j])
        }()
    }

    // context.WithTimeout does the following
    /*
       time.Sleep(time.Second)
       cancel()
    */

    sum := 0
    for _, ch := range sumChs {
        sum += <-ch
    }
    fmt.Println("Sum: ", sum)
}
```

context.Background() creates a singleton context.Context. We will need it as the base context to create a cancellable context.

```
type queue chan int

func (q queue) try_enqueue(num int) bool {
    select {
    case q <- num:
        return true
    default:
    }
    return false
}

func (q queue) try_dequeue() (int, bool) {
    select {
    case num, ok := <-q:
        return num, ok
    default:
    }
    return 0, false
}

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
        }
    }
}

func consumer(ctx context.Context, q queue, sumCh chan<- int) {
    sum := 0
    for {
        select {
        case <-ctx.Done():
            sumCh <- sum
            close(sumCh)
            return
        case num := <-q:
            sum += num
        }
    }
}
```

In the producer and consumer, the done case in select is replaced with case <-ctx.Done(). ctx.Done() returns a channel that is closed when the cancel function is called. NOTE: We can manually cancel() the context earlier if we desire. **Double cancel()ling a context does not result in a panic.**

L7 - Concurrency Patterns in Go

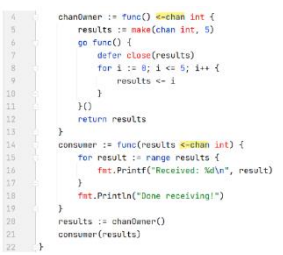
Patterns
- Separation of concerns
  - Data chunks (confinement), Error handling, Data processing (pipeline)

Confinement
- Achieve safe operation
  - Synchronization primitives for sharing memory (e.g., sync.Mutex)
  - Synchronization via communicating (e.g., channels)
- Safe concurrency with good performance
  - Immutable data
  - Data protected by confinement

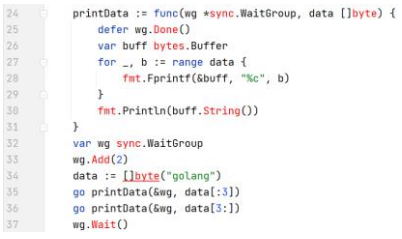
Achieving confinement
- Ad-hoc confinement
  - By convention, data is modified only from one goroutine, even though it is accessible from multiple goroutines
  - Needs some static analysis to ensure safety
- Lexical confinement
  - Restrict the access to shared locations

Lexical Confinement

Line 4 & 14: expose only the reading/writing handle of the channel.



Line 35-36: Expose only a slice of the array.



The for-select loop

- Context
  - Sending iteration variables out on a channel
  - Looping and waiting to be stopped

```
for { // Either loop infinitely or
range over something
  select {
  // Do some work with channels
  }
}
```

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

- Loop
- Keeps the select statement as short as possible
- Do work while done channel is not closed

Preventing goroutines from leaking

- Goroutine do cost resources!
- Ensure termination of your goroutines
  - When it has completed its work
  - When it cannot continue its work due to an unrecoverable error
  - When it's told to stop working
- Convention: if a goroutine is responsible for creating a goroutine, it is also responsible for ensuring it can stop the goroutine

Leaking goroutines

```
4   doWork := func(strings <-chan string) <-chan interface{} {
5       completed := make(chan interface{})
6       go func() {
7           defer fmt.Println("doWork exited.")
8           defer close(completed)
9           for s := range strings {
10              // Do something interesting
11              fmt.Println(s)
12          }
13      }()
14      return completed
15  }
16  doWork(nil)
17  // Perhaps more work is done here
18  fmt.Println("Done.")
```

Line 16: goroutines will accumulate in memory
Leaking because

- a nil channel was passed and
- Line 9: forever blocks due to semantics of reading from a nil channel
- read from a NIL channel → BLOCKED

Stopping reader goroutines (1)

```
24      done := make(chan interface{})
25      terminated := doWork(done, nil)
26      go func() {
27          // Cancel the operation after 1 second.
28          time.Sleep(1 * time.Second)
29          fmt.Println("Canceling doWork goroutine...")
30          close(done)
31      }()
32      <-terminated
33      fmt.Println("Done.")
```

- Line 25: done channel passed to the doWork function
- Line 26: another goroutine will cancel the goroutine spawned from doWork if more than one second passes
- Line 32: join the goroutine spawned from doWorkwith the main goroutine

Stopping reader goroutines (2)

Line 12-19: for-select pattern in use

```
4   doWork := func(
5       done <-chan interface{},
6       strings <-chan string,
7   ) <-chan interface{} {
8       terminated := make(chan interface{})
9       go func() {
10          defer fmt.Println("doWork exited."
11          defer close(terminated)
12          for {
13              select {
14              case s := <-strings:
15                  // Do something interestin
16                  fmt.Println(s)
17              case <-done:
18                  return
19              }
20          }
21      }()
22      return terminated
23  }
```

enter here when done chan is closed

Stopping writer goroutines (2)

```
4   newRandStream := func(done <-chan interface{}) <-chan int {
5       randStream := make(chan int)
6       go func() {
7           defer fmt.Println("newRandStream closure exited.")
8           defer close(randStream)
9           for {
10              select {
11              case randStream <- rand.Int():
12              case <-done:
13                  return
14              }
15          }
16      }()
17      return randStream
18  }
```

```
19      done := make(chan interface{})
20      randStream := newRandStream(done)
21      fmt.Println("3 random ints:")
22      for i := 1; i <= 3; i++ {
23          fmt.Printf("%d: %d\n", i, <-randStream)
24      }
25      close(done)
26      // Simulate ongoing work
27      time.Sleep(1 * time.Second)
28  }
```

CS3211

Error handling

Goal: gracefully handle erroneous states

- Responsibility for handling errors
  - A goroutine to maintain complete information about the state of the program
  - All goroutines send their errors to the state-goroutine that can make an informed decision about what to do
  - Couple the potential result with the potential error
    - Errors should be tightly coupled with your result type, and passed along through the same lines of communication

Error handling example

```
4   type Result struct {
5       Error error
6       Response *http.Response
7   }
```

```
8   checkStatus := func(done <-chan interface{}, urls ...string) <-chan Result {
9       results := make(chan Result)
10      go func() {
11          defer close(results)
12          for _, url := range urls {
13              var result Result
14              resp, err := http.Get(url)
15              result = Result{Error: err, Response: resp}
16              select {
17              case <-done:
18                  return
19              case results <- result:
20              }
21          }
22      }()
23      return results
24  }
```

```
25      done := make(chan interface{})
26      defer close(done)
27      urls := []string{"https://www.google.com", "https://badhost"}
28      for result := range checkStatus(done, urls...) {
29          if result.Error != nil {
30              fmt.Printf("error: %v", result.Error)
31              continue
32          }
33          fmt.Printf("Response: %v\n", result.Response.Status)
34      }
```

```
36      done := make(chan interface{})
37      defer close(done)
38      errCount := 0
39      urls := []string{"a", "https://www.google.com", "b", "c", "d"}
40      for result := range checkStatus(done, urls...) {
41          if result.Error != nil {
42              fmt.Printf("error: %v\n", result.Error)
43              errCount++
44              if errCount >= 3 {
45                  fmt.Println("Too many errors, breaking!")
46                  break
47              }
48              continue
49          }
50          fmt.Printf("Response: %v\n", result.Response.Status)
51      }
```

Pipeline

- Multiple types
  - Instruction pipelines
  - Graphics pipelines
  - Software pipelines
- A set of data processing elements connected in series, where the output of one element is the input of the next one
  - Stages
  - Connect the stages

Pipelines in Go

- A series of stages connected by channels
  - Each stage is a group of goroutines running the same function
- In each stage, the goroutines
  - receive values from upstream via inbound channels
  - perform some function on that data, usually producing new values
  - send values downstream via outbound channels
- Each stage has any number of inbound and outbound channels, except the first and last stages
  - The first stage: source or producer
  - The last stage: the sink or consumer

Pipeline pattern in concurrent programming

Separate the concerns of each stage

- Modify stages independently of one another,
- Mix and match how stages are combined independent of modifying the stages
- Process each stage concurrent to upstream or downstream stages
- Fan-out, or rate-limit portions of your pipeline

Pipelines design

For efficiency

- Designer should divide the work and resources among the stages such that they all take the same time to complete their tasks
- Fan-out to decrease the processing time for a stage if that is the bottleneck
- Use of I/O and multiple CPUs for processing streams of data

Not obviously faster than a task pool

- Tweaking is needed to make the pipeline more efficient than a task pool
- Pipeline is better if there is a cap on a specific resource that is needed by all tasks in the task pool (at different times)
- For example, reading or writing to a restricted network link

Pipeline Example

```
generator := func(done <-chan interface{},
    integers ...int
) <-chan int {
    intStream := make(chan int)
    go func() {
        defer close(intStream)
        for _, i := range integers {
            select {
            case <-done:
                return
            case intStream <- i:
            }
        }
    }()
    return intStream
}
```

```
multiply := func(
    done <-chan interface{},
    intStream <-chan int,
    multiplier int,
) <-chan int {
    multipliedStream := make(chan int)
    go func() {
        defer close(multipliedStream)
        for i := range intStream {
            select {
            case <-done:
                return
            case multipliedStream <- i*multiplier:
            }
        }
    }()
    return multipliedStream
}
```

Pipeline Example

```
add := func(
    done <-chan interface{},
    intStream <-chan int,
    additive int,
) <-chan int {
    addedStream := make(chan int)
    go func() {
        defer close(addedStream)
        for i := range intStream {
            select {
            case <-done:
                return
            case addedStream <- i+additive:
            }
        }
    }()
    return addedStream
}
```

```
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)
}
```

L7S22/23

Iteration	Generator	Multiply	Add	Multiply	Value
0	1				
0		1			
0	2		2		
0		2		3	
1	3		4		6
close(done)	(closed)	3		5	
		(closed)	6		
			(closed)	7	
				(closed)	
					(exit range)

Fan-out, fan-in pattern

- Problem: stages in a pipeline might be slower than the other and they might benefit from parallelism
  - Computationally intensive work
- **Fan-out**: start multiple goroutines to handle input from the pipeline
- **Fan-in**: combine multiple results into one channel

Fan-out

Fan-out a stages of the processing if

- It doesn't rely on values that the stage had calculated before
- It takes a long time to run

No guarantee on the order concurrent copies run, nor in what order they return

- A naive implementation of fan-out only works if the order in which results arrive is unimportant

```
numFinders := runtime.NumCPU()
finders := make([]<-chan int, numFinders)
for i := 0; i < numFinders; i++ {
    finders[i] = primeFinder(done, randIntStream)
}
```

randIntStream is split among several channels.

Fan-out design

- Number of goroutines that are spined up matters
  - Use **runtime.NumCPU()** to find the number of OS threads that are used to run the goroutines (max concurrency we can get)
  - As a rule of thumb, fan-out runtime.NumCPU() goroutines, or profile your code to enhance the performance

Fan-in

- Involves *multiplexing* or *joining* together multiple streams of data into a single stream (merging)
  - Consumers read from the multiplexed channel
  - Spin up one goroutine for each incoming channel, and transfer the information from the multiple streams into the multiplexed stream

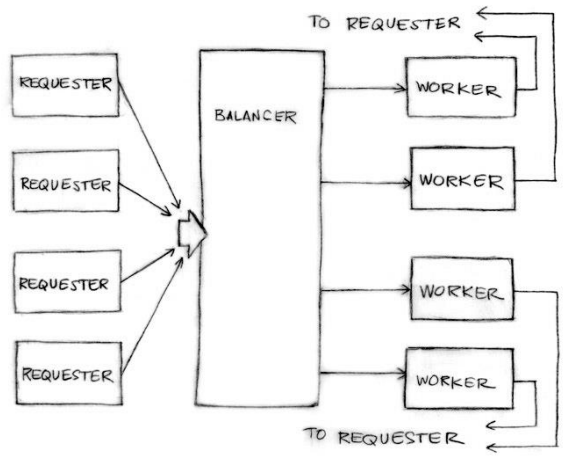
```
fanIn := func(
    done <-chan interface{},
    channels ...<-chan interface{},
) <-chan interface{} {
    var wg sync.WaitGroup
    multiplexedStream := make(chan interface{})
    multiplex := func(c <-chan interface{}) {
        defer wg.Done()
        for i := range c {
            select {
            case <-done:
                return
            case multiplexedStream <- i:
            }
        }
    }
    // Select from all the channels
    wg.Add(len(channels))
    for _, c := range channels {
        go multiplex(c)
    }
    // Wait for all the reads to complete
    go func() {
        wg.Wait()
        close(multiplexedStream)
    }()
    return multiplexedStream
}
```

Fan-out, fan-in example

- Fan out to find prime numbers
- Fan in to print the results
- A bit too many goroutines for such a simple problem

```
done := make(chan interface{})
defer close(done)
start := time.Now()
rand := func() interface{} { return rand.Intn(50000000) }
randIntStream := toInt(done, repeatFn(done, rand))
numFinders := runtime.NumCPU()
fmt.Printf("Spinning up %d prime finders.\n", numFinders)
finders := make([]<-chan interface{}, numFinders)
fmt.Println("Primes:")
for i := 0; i < numFinders; i++ {
    finders[i] = primeFinder(done, randIntStream)
}
for prime := range take(done, fanIn(done, finders...), 10) {
    fmt.Printf("\t%d\n", prime)
}
fmt.Printf("Search took: %v", time.Since(start))
```

A realistic load balancer (1)



- The **requesters** send Requests to the balancer
  - return channel inside the request

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

func requester(work chan<- Request) {
    c := make(chan int)
    for {
        // Kill some time (fake load).
        Sleep(rand.Int63n(nWorker * 2 * Second))
        work <- Request{workFn, c} // send request
        result := <-c              // wait for answer
        furtherProcess(result)
    }
}
```

Worker definition
- Channel of requests
- Include load tracking data
- If worker is less overloaded, it will be at top of heap

```
type Worker struct {
    requests chan Request // work to do (buffered channel)
    pending  int          // count of pending tasks
    index     int         // index in the heap
}
```

Worker
- The channel of requests (w.requests) delivers requests to each worker
- The balancer tracks the number of pending requests as a measure of load
- Each response goes directly to its requester

```
func (w *Worker) work(done chan *Worker) {
    for {
        req := <-w.requests // get Request from balancer
        req.c <- req.fn()   // call fn and send result
        done <- w           // we've finished this request
    }
}
```

Balancer needs
- A pool of workers
- A single channel to which requesters can report task completion

```
type Pool []*Worker

type Balancer struct {
    pool Pool
    done chan *Worker
}
```
// only 1 done chan per balancer

Balancer
- **Dispatches**
- **Completes**

```
func (b *Balancer) balance(work chan Request) {
    for {
        select {
        case req := <-work: // received a Request...
            b.dispatch(req) // ...so send it to a Worker
        case w := <-b.done: // a worker has finished ...
            b.completed(w)  // ...so update its info
        }
    }
}
```
pipeline
1. read from work chan
2. put stuff into heap n process
3. put into worker queue

A **heap of channels** –the **Pool**
- An implementation of the Heap interface
- Balance by making the Pool a heap tracked by load

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

Dispatch

```
// Send Request to worker
func (b *Balancer) dispatch(req Request) {
    // Grab the least loaded worker...
    w := heap.Pop(&b.pool).(*Worker)
    // ...send it the task.
    w.requests <- req
    // One more in its work queue.
    w.pending++
    // Put it into its place on the heap.
    heap.Push(&b.pool, w)
}
```

Completed

```
// Job is complete; update heap
func (b *Balancer) completed(w *Worker) {
    // One fewer in the queue.
    w.pending--
    // Remove it from heap.
    heap.Remove(&b.pool, w.index)
    // Put it into its place on the heap.
    heap.Push(&b.pool, w)
}
```

- A complex problem can be broken down into easy-to-understand components
- The pieces can be composed concurrently
- The result is easy to understand, efficient, scalable, and correct
- Decomposition allows for parallelism

Discussion
How to enhance the load balancer?
Will more concurrency translate into more performance?:
What patterns can we use in this problem?

- Performance might decrease, we don't know
- Let other workers steal work from other quques
- try to take the very last work
- Balancer still sequential
- Can perform pipelining
- Multiple req in different stages of the pipeline
- Patterns: for-select, fan-out, error handling

Summary
We discussed patterns specific to Go
- Enabled by channels and lightweight goroutines
- Tradeoffs of using many goroutines/channels
- Splitting the work helps us achieve more parallelism, but can be slower at times

---

Tutorial 6: Advanced Go concurrency patterns

Exit conditions
Recall the use of done channels in the last tutorial: a mechanism to signal to all the goroutines that they should exit due to some reason. Typically, some set of pre-defined exit conditions have been met, and you want the goroutines to free up the resources acquired (e.g. memory, file descriptor, DB connection.)

Typical cases are
- The spawner of the goroutines receives an external signal to terminate.
- There are two dependant goroutines that are running concurrently but one fails / is cancelled.
- Some timeout has been reached / a deadline has passed (e.g., deadline while waiting for some response).

```
package main
import (
    "context"
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"
)

// Returns a `sigs` channel that blocks until either the program sees
either signal
func handleSigs() chan struct{} {
    done := make(chan struct{})
    go func() {
        sigs := make(chan os.Signal, 1)
        signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
        <-sigs
        close(done)
    }()
    return done
}

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

func main() {
    var wg sync.WaitGroup
    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()
        time.Sleep(time.Millisecond * 100)
        fmt.Printf("ctx3 done at %v\n", time.Now().Sub(startTime))
        wg.Done()
    }()

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

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

In this example, we have built a context tree.
ctx (timeout at 4s)
|
|------> ctx3 (cancellable like the other contexts, but we expose the cancel() function to the program)
|
v
ctx2 (timeout at 2s)

ctx2 done at 2.001188291s
ctx3 done at 4.0010545s
ctx done at 4.001064083s

go run demo1.go
ctx2 done at 2.001142625s
^Cctx3 done at 2.515675042s
signal in at 2.515656875s
ctx 4.001095084s

go run demo1.go
^Cctx3 done at 1.829278834s
signal in at 1.829256292s
ctx2 done at 2.001535459s
ctx 4.001105834s

We can see that cancellation of ctx3 also does not affect ctx and ctx2.

Fan-out, fan-in
A classic concurrency pattern in Go is fan-out, fan-in, where a stream (channel) of data is multiplexed to a set of goroutines to be processed, and merged back into one stream (channel).

```
package main
import (
  "fmt"
  "math/rand"
  "sync"
  "time"
)

// Representing each individual event
type Event struct {
  id       int64
  procTime time.Duration
}

type EventFunc func(Event) Event
type worker struct {
  inputCh  <-chan Event
  outputCh chan<- Event
}

func newWorker(inputCh, outputCh chan Event) *worker {
  return &worker{
    inputCh:  inputCh,
    outputCh: outputCh,
  }
}

func (w *worker) start(
  done <-chan struct{},
  fn EventFunc, wg *sync.WaitGroup,
) {
  go func() {
    defer wg.Done()
    for {
      select {
      case e, more := <-w.inputCh:
        if !more {
          return
        }
        select {
        case w.outputCh <- fn(e):
        case <-done:
          return
        }
      case <-done:
        return
      }
    }
  }()
}
```

```
// Randomly generates events and pushes them into an output channel
`outputCh`
func genEventsCh() chan Event {
  outputCh := make(chan Event)
  go func() {
    counter := int64(1)
    rand.Seed(time.Now().Unix())
    for i := 0; i < 30; i++ {
      outputCh <- Event{
        id:       counter,
        procTime: time.Duration(rand.Intn(100)) * time.Millisecond,
      }
      counter++
    }
    close(outputCh)
  }()
  return outputCh
}

func main() {
  done := make(chan struct{})
  outputCh := make(chan Event, 1)

  inputCh := genEventsCh()

  // Fan-out the stream of input to multiple workers
  var wg sync.WaitGroup
  for i := 0; i < 10; i++ {
    wg.Add(1)
    newWorker(inputCh, outputCh).
      start(done, func(e Event) Event {
        time.Sleep(e.procTime)
        return e
      }, &wg)
  }

  readerDone := make(chan struct{})
  go func() {
    for output := range outputCh {
      fmt.Printf("Event id: %d\n", output.id)
    }
    close(readerDone)
  }()

  wg.Wait()

  // Close outputCh and wait for reader to finish reading
  close(outputCh)
  <-readerDone
}
```

Task 1 : Serializing the Event output
Solution 1: Sort outputs
- Print after all outputs only
- Not very scalable/performant

```
readerDone := make(chan struct{})
go func() {
  var s []int64
  for output := range outputCh {
      s = append(s,output.id)
  }
  for id := range s {
      fmt.Printf("Event id: %d\n", id)
  }
  close(readerDone)
}()
```

Solution 2: "Reorder" Buffer
- Print events that arrive in-order
- Store events that are out of order, print them later as necesssary

Higher order channels
The problem of the reordering of the Event-s stems from the variable processing time and that the workers are not aware of each other's order completion. We can borrow the concept of Promise from other languages to fix this.

The recipe:
- A channel of workers queuing for work
- A channel of promises of Event processed

```
package main

import (
  "fmt"
  "math/rand"
  "sync"
  "time"
)

type Event struct {
  id       int64
  procTime time.Duration
}

type EventFunc func(Event) Event

type worker struct {
  inputCh  chan Event
  outputCh chan Event
}

func newWorker() *worker {
```

```
  return &worker{
    inputCh:  make(chan Event),
    outputCh: make(chan Event),
  }
}

func (w *worker) start(
  done chan struct{},
  fn EventFunc, workerQueue chan *worker, wg *sync.WaitGroup,
) {
  go func() {
    defer func() {
      close(w.outputCh)
      wg.Done()
    }()

    for {
      select {
      case workerQueue <- w: // <-- change: sign up for work
        select {
        case e := <-w.inputCh:
          select {
          case w.outputCh <- fn(e):
          }
        }
      case <-done:
        return
      }
    }
  }()
}

func orderedMux(
  done chan struct{},
  inputCh chan Event, workerQueue chan *worker, workerOutputCh chan chan
Event,
) {
  go func() {
    for {
      select {
      case e, more := <-inputCh:
        if !more {
                close(done)
          return
        }
        select {
        case w := <-workerQueue:
          select {
          case workerOutputCh <- w.outputCh:
          case <-done:
            return
          }
          w.inputCh <- e
        case <-done:
        }
      case <-done:
        return
      }
    }
  }()
}

func genEventsCh() chan Event {
  outputCh := make(chan Event)
  go func() {
    counter := int64(1)
    rand.Seed(time.Now().Unix())
    for i := 0; i < 30; i++ {
      outputCh <- Event{
        id:       counter,
        procTime: time.Duration(rand.Intn(100)) * time.Millisecond,
      }
      counter++
    }
    close(outputCh)
  }()
  return outputCh
}

func main() {
    numWorkers := 10;
  done := make(chan struct{})
  workerQueue := make(chan *worker)
  workerOutputCh := make(chan chan Event, numWorkers)

  inputCh := genEventsCh()

  orderedMux(done, inputCh, workerQueue, workerOutputCh)

  var wg sync.WaitGroup
  for i := 0; i < numWorkers; i++ {
    wg.Add(1)
    newWorker().start(done, func(e Event) Event {
      time.Sleep(e.procTime)
      return e
    }, workerQueue, &wg)
  }

  readerDone := make(chan struct{})
  go func() {
    for outputCh := range workerOutputCh { // <-- reading a stream of
                                           promises
      output, more := <-outputCh // <-- read from the promise once only
      if !more {
        break // <-- a worker breaks its promise; workers exiting
      }
        fmt.Printf("Event id: %d\n", output.id)
    }
    close(readerDone)
  }()

  //time.Sleep(2 * time.Second)

  // stop all workers and wait for them to exit
  wg.Wait()

  close(workerOutputCh)
  <-readerDone
}
```

We use a multiplexer goroutine orderedMux to hijack the inputCh.

orderedMux will
- read Event e from inputCh,
- assign it to any ready worker in workerQueue, and then
- send the worker-'s outputCh to workerOutputCh.

Notice that the w.outputCh sent basically acts as a promise of a result that can be read later on.

Now, main will read from the channel of "promises" instead. We use for-range here to read out the output channels queued in workerOutputCh. For each of them, we will read once to get the output. If a promise is broken, we know that the worker must have exited. Since we want a stream of serial Event-s, we choose to break from the reader loop without reading the remaining promises.

Pipelining

A typical client-server: Suppose we have a server serving a constant stream of requests, and each request is served in some number of steps.

```
package main

import (
    "fmt"
    "math/rand"
    "sync/atomic"
    "time"
)

var array = make([]int, 10000000, 10000000)

func populate(array []int) {
    rand.Seed(time.Now().Unix())
    for i := 0; i < len(array); i++ {
        array[i] = rand.Int()
    }
}

func busyForLoops(loops int) {
    sum := 0
    for num := range array[:loops] {
        sum += num
    }
}

type doneCh chan struct{}

type Request struct {
    id int64
}

var (
    reqId         int64 = 0
    completedReqs int64 = 0
)

func NewReq() *Request {
    for {
        id := atomic.LoadInt64(&reqId)
        if atomic.CompareAndSwapInt64(&reqId, id, id+1) {
            return &Request{id + 1}
        }
    }
}

func (r *Request) Done() {
    atomic.AddInt64(&completedReqs, 1)
    fmt.Printf("req %v\n", r.id)
}

func processReqStep1(req *Request) *Request {
    busyForLoops(2000000)
    return req
}

func processReqStep2(req *Request) *Request {
    busyForLoops(5000000)
    return req
}

func processReqStep3(req *Request) *Request {
    busyForLoops(10000000)
    return req
}

func processReqStep4(req *Request) *Request {
    busyForLoops(2000000)
    return req
}
```

```
func toPipelineStage(
    done doneCh, reqCh <-chan Request, instances int, fn func(*Request)
*Request,
) chan Request {
    outCh := make(chan Request)
    for i := 0; i < instances; i++ {
        go func() {
            for req := range reqCh {
                select {
                case outCh <- *fn(&req):
                case <-done:
                    return
                }
            }
        }()
    }
    return outCh
}

func client(done doneCh, reqCh chan<- Request) {
    for {
        select {
        case reqCh <- *NewReq():
        case <-done:
            return
        }
    }
}

func server(done doneCh, reqCh <-chan Request) {
    for {
        select {
        case completed := <-reqCh:
            completed.Done()
        case <-done:
            return
        }
    }
}
const (
    numClients = 2
    numServers = 5
)
func init() {
    populate(array)
}

func main() {
    start, done := make(doneCh), make(doneCh)
    reqCh := make(chan Request, 100)
    // Make pipeline of 4 stages
    outCh := toPipelineStage(done, reqCh, 2, processReqStep1)
    outCh = toPipelineStage(done, outCh, 5, processReqStep2)
    outCh = toPipelineStage(done, outCh, 10, processReqStep3)
    outCh = toPipelineStage(done, outCh, 5, processReqStep4)

    // Spawn a client and multiple servers
    for i := 0; i < numClients; i++ {
        go func() {
            client(done, reqCh)
        }()
    }
    for i := 0; i < numServers; i++ {
        go func() {
            <-start
            server(done, outCh)
        }()
    }
    close(start)
    time.Sleep(2 * time.Second)
    close(done)

    time.Sleep(time.Second) // Wait for all to exit
    fmt.Printf("# of completed reqs: %v\n",
atomic.LoadInt64(&completedReqs))
}
```

Pipelining seems to be worse than the classic request-to-goroutine model. So why are we doing this?

Imagine that:

- Stage 1 makes requests to some remote service that only allow 15 connections at the same time
- Stage 2 runs some variable CPU-bound workload whose runtime is significantly longer than other stages combined
- Stage 3 consumes large amount of memory, and the memory on the machine can only accommodate 10 such instances

The bottleneck appears in stage 3, and we cannot accommodate more than 10 concurrent requests. Should we only use a worker pool of 10 goroutines to prevent any OOM failure?

Pipelining model provides a nicer way to scale each stage, should there be any resource constraints.

In this case, we can simply make 15 instances of stage 1, 10 instances of stage 3, and tune the number of instances in stage 2 so that we achieve a good CPU utilization rate for a constant stream of requests.

This allows us to achieve better concurrency than having to control the number of workers for requests.

L8 - Classical Synchronization Problems in C++ and Go

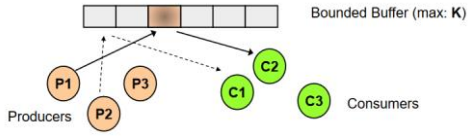
Motivation

Classical synchronization problems model problems that we have nowadays in our computer systems

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.
H20	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: specification

- Processes share a buffer (bounded or unbounded)
  - Producers produce items to insert in buffer
    - Only when the buffer is not full (< K items)
- Consumers remove items from buffer
  - Only when the buffer is not empty (> 0 items)



Producer consumer: buffered version


```
while (TRUE) {
        Produce Item;

        wait(notFull);
        wait(mutex);
        buffer[in] = item;
        in = (in+1) % K;
        count++;
        signal(mutex);
        signal(notEmpty);
}
```

Producer

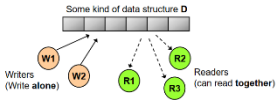
```
while (TRUE) {

        wait(notEmpty);
        wait(mutex);
        item = buffer[out];
        out = (out+1) % K;
        count--;
        signal(mutex);
        signal(notFull);

        Consume Item;
}
```

Consumer

notFull = K, notEmpty = 0

Readers writers: specification

- Processes share a data structure D:
  - Reader: retrieves information from D
  - Writer: modifies information in D
- Writer must have **exclusive** access to D
- Reader can access with other readers



Writers

```
roomEmpty.wait ()
    #critical section for writers
roomEmpty.signal ()
```

Readers

```
mutex.wait ()
readers += 1
if readers == 1:
    roomEmpty.wait () # first in locks
mutex.signal ()
# critical section for readers
mutex.wait ()
readers -= 1
if readers == 0:
    roomEmpty.signal () # last out unlocks
mutex.signal ()
```

Starvation of writers is possible

Lightswitch definition

Readers writers with light switch

Writers

```
roomEmpty.wait ()
    #critical section for writers
roomEmpty.signal ()
```

Readers

```
readLightswitch.lock(roomEmpty)
    # critical section
readLightswitch.unlock(roomEmpty)
```

#starving writers

Use a

turnstile = Semaphore (1)

Lightswitch :

```
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 ()
```

No-starve readers writers

Writers

```
turnstile.wait ()
roomEmpty.wait ()
    # critical section for writers
turnstile.signal ()
roomEmpty.signal ()
```

Readers

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

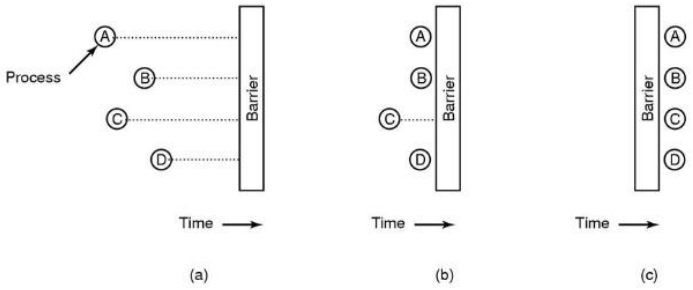
Readers writers locks

- Programming languages have rwlocks
  - C++17 onwards: use shared_mutex
  - Go has RWLock

```cpp
class ThreadSafeCounter {
 public:
  ThreadSafeCounter() = default;

  // Multiple threads/readers can read
  //the counter's value at the same time.
  unsigned int get() const {
    std::shared_lock lock(mutex_);
    return value_;
  }

  // Only one thread/writer can
  // increment/write the counter's value.
  unsigned int increment() {
    std::unique_lock lock(mutex_);
    return ++value_;
  }

  // Only one thread/writer can reset/write
  //the counter's value.
  void reset() {
    std::unique_lock lock(mutex_);
    value_ = 0;
  }

 private:
  mutable std::shared_mutex mutex_;
  unsigned int value_ = 0;
};
```

Barrier: specification

- Any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier



Barrier usage

- Appears in many collective routines as part of directive-based parallel languages
  - Parallel for loop in OpenMP
  - Collective communication in MPI
- Part of the programming language
  - std::barrier in C++20
  - WaitGroup wait in Go

Types of barriers

- Single use barrier or latch (std::latch in C++20)
  - Starts in the raised state and cannot be re-raised once it is in the lowered state
- Reusable barriers (std::barrier in C++20)
  - Once the arriving threads are unblocked from a barrier phase's synchronization point, the same barrier can be reused
  - Combining tree barrier - a hierarchical way of implementing barrier
    - Resolve the scalability by avoiding the case that all threads are spinning at the same location

C++: Barrier implementation (2)

```cpp
struct Barrier2 {
  std::ptrdiff_t expected;
  std::ptrdiff_t count;
  std::mutex mut;
  std::counting_semaphore<> turnstile;
  std::counting_semaphore<> turnstile2;

  Barrier2(std::ptrdiff_t expected)
      : expected{expected}, count{0}, mut{},
        turnstile{0}, turnstile2{1} {}

  void arrive_and_wait() {
    {
      std::scoped_lock lock{mut};
      count++;
      if (count == expected) {
        // Close waiter turnstile
        turnstile2.acquire();
        // Open turnstile into the critical section
        turnstile.release();
      }
    }
    turnstile.acquire();
    turnstile.release();
```

- Use 2 turnstiles: raise and lower the barrier

```cpp
    {
      std::scoped_lock lock{mut};
      count--;
      if (count == 0) {
        // Close turnstile to reset barrier
        turnstile.acquire();
        // Open second turnstile to let waiters through
        turnstile2.release();
      }
    }
    turnstile2.acquire();
    turnstile2.release();
  }
};
```


C++: Barrier implementation (3)

```cpp
// Use a preloaded turnstile to let threads through faster
struct Barrier3 {
  std::ptrdiff_t expected;
  std::ptrdiff_t count;
  std::mutex mut;
  std::counting_semaphore<> turnstile;
  std::counting_semaphore<> turnstile2;

  Barrier3(std::ptrdiff_t expected)
      : expected{expected}, count{0}, mut{},
        turnstile{0}, turnstile2{1} {}

  void arrive_and_wait() {
    {
      std::scoped_lock lock{mut};
      count++;
      if (count == expected) {
        // Close waiter turnstile
        turnstile2.acquire();
        // Open turnstile into the critical section
        turnstile.release(expected);
      }
    }
    turnstile.acquire();
```

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

```cpp
    {
      std::scoped_lock lock{mut};
      count--;
      if (count == 0) {
        // Close turnstile to reset barrier
        turnstile.acquire();
        // Open second turnstile to let waiters through
        turnstile2.release(expected);
      }
    }

    turnstile2.acquire();
  }
};
```

Go: Barrier implementation

- Use 2 WaitGroups for a reusable barrier for a group of goroutines

```go
type Barrier1 struct {
    wg  sync.WaitGroup
    wg2 sync.WaitGroup
}

func (b *Barrier1) Init(expected int) {
    b.wg.Add(expected)
    b.wg2.Add(expected)
}

func (b *Barrier1) Wait() {
    b.wg.Done()
    b.wg.Wait()
    // This line only reached when expected
    //threads have called Wait
    // Reset the barrier now
    b.wg.Add(1)
    b.wg2.Done()
    // Wait because the barrier might not
    //be fully reset yet
    b.wg2.Wait()
    // Now barrier is fully reset
    b.wg2.Add(1)
}
```

C++: `std::barrier`

- In C++, create a synchronization point
  - Lines 21 & 25: wait for all threads to arrive at the barrier

```cpp
const auto tas = { "hw", "g", "zr", "c" };

auto on_completion = []() noexcept {
  // locking not needed here
  static auto phase = "... done\n" "Cleaning up...\n";
  std::cout << phase;
  phase = "... done\n";
};
std::barrier sync_point(std::ssize(tas), on_completion);

auto work = [&](std::string name) {
  std::string product = "  " + name + " worked\n";
  std::cout << product;
  sync_point.arrive_and_wait();

  product = "  " + name + " cleaned\n";
  std::cout << product;
  sync_point.arrive_and_wait();
};

std::cout << "Starting...\n";
std::vector<std::thread> threads;
for (auto const& worker : tas) {
  threads.emplace_back(work, worker);
}
for (auto& thread : threads) {
  thread.join();
}
```

C++: Barrier implementation
Attempt 1: Threads can go ahead other threads by one lap

```cpp
struct BarrierAttempt1 {
  std::ptrdiff_t expected;
  std::ptrdiff_t count;
  std::mutex mut;
  std::counting_semaphore<> turnstile;

  BarrierAttempt1(std::ptrdiff_t expected)
    : expected{expected}, count{0}, mut{}, turnstile{0}

  void arrive_and_wait() {
    {
      std::scoped_lock lock{mut};
      count++;
      if (count == expected) {
        // Open turnstile
        turnstile.release();
      }
    }

    turnstile.acquire();
    turnstile.release();

    {
      std::scoped_lock lock{mut};
      count--;
      if (count == 0) {
        // Close turnstile to reset barrier
        turnstile.acquire();
      }
    }
  }
};
```

<u>Dining philosophers: specification</u>

- Appeared when computers were competing for access to tape drive peripherals
- Models the problem of allocating limited resources to a group of processes in a deadlock-free and starvation-free manner
- An algorithm that solves this problem without deadlock
  - Low contention: performs wonderfully when the philosophers spend any appreciable amount of time thinking, compared to eating
  - High contention: philosophers are hungry

Attempt 1

Deadlock: All philosophers simultaneously pick up the left chopstick, none can proceed

Fix the attempt: Make the philosphers put down left chopstick if right chopstick cannot be acquired -> No deadlock: but livelock: all philosophers take up, put down, repeat…

```
#define N 5
#define LEFT i
#define RIGHT ((i+1) % N)

//For philosopher i
while (TRUE){

      Think();
      //hungry, need food!
      takeChpStick(LEFT);
      takeChpStick(RIGHT);

      Eat();

      putChpStick(LEFT);
      putChpStick(RIGHT);
}
```

Attempt 2

Mutex makes it sequential!

```
#define N 5
#define LEFT i
#define RIGHT ((i+1) % N)

//For philosopher i
while (TRUE){
      Think( );

      wait(mutex);

      takeChpStick(LEFT);
      takeChpStick(RIGHT);
      Eat( );
      putChpStick(LEFT);
      putChpStick(RIGHT);

      signal(mutex);
}
```

C++: Dining philosophers' implementation (1)

- Uses *some* deadlock avoidance algorithm
  - The objects are locked by an unspecified series of calls to lock, try_lock, and unlock. If a call to lock or unlock results in an exception, unlock is called for any locked objects before rethrowing
  - Livelock can still happen

```cpp
// Basic solution to the dining philosopher's problem
template <size_t NumP>
struct DiningTable1 {
  using ChpStick = std::mutex;

  ChpStick chpSticks[NumP];

  // pid = philosopher id
  ChpStick& get_left_chpStick(size_t pid) { return chpSticks[pid]; }
  ChpStick& get_right_chpStick(size_t pid) { return chpSticks[(pid + 1) % NumP]; }

  void eat(size_t pid, void (*eat_callback)(size_t pid)) {
    std::scoped_lock lock{get_left_chpStick(pid), get_right_chpStick(pid)};
    eat_callback(pid);
  }
};
```

Go: Dining philosophers' implementation

- Use odd-even ring communication to avoid the deadlock

```go
type ChpStick struct{}

type DiningTable1 struct {
    numPhilosophers int
    chpStickChs         []chan ChpStick
}

func (t *DiningTable1) Init(numPhilosophers int) {
    t.numPhilosophers = numPhilosophers

    t.chpStickChs = make([]chan ChpStick, 0, numPhilosophers)
    for i := 0; i < numPhilosophers; i++ {
        chpStick := make(chan ChpStick, 1)
        chpStick <- ChpStick{}
        t.chpStickChs = append(t.chpStickChs, chpStick)
    }
}

func (t *DiningTable1) leftChpStickCh(pid int) chan ChpStick {
    return t.chpStickChs[pid]
}
func (t *DiningTable1) rightChpStickCh(pid int) chan ChpStick {
    return t.chpStickChs[(pid+1)%t.numPhilosophers]
}
func (t *DiningTable1) evenChpStickCh(pid int) chan ChpStick {
    if pid%2 == 0 {
        return t.leftChpStickCh(pid)
    } else {
        return t.rightChpStickCh(pid)
    }
}
func (t *DiningTable1) oddChpStickCh(pid int) chan ChpStick {
    if pid%2 == 1 {
        return t.leftChpStickCh(pid)
    } else {
        return t.rightChpStickCh(pid)
    }
}

func (t *DiningTable1) Eat(pid int, eat_callback func(pid int)) {
    evenChpStickCh := t.evenChpStickCh(pid)
    oddChpStickCh := t.oddChpStickCh(pid)

    // Use even / odd chpSticks so the resulting
    // chpStick locking order is acyclic
    <-evenChpStickCh
    <-oddChpStickCh

    eat_callback(pid)

    evenChpStickCh <- ChpStick{}
    oddChpStickCh <- ChpStick{}
}

func (t *DiningTable2) Init(numPhilosophers int) {
    t.numPhilosophers = numPhilosophers

    t.chpStickChs = make([]chan ChpStick, 0, numPhilosophers)
    for i := 0; i < numPhilosophers; i++ {
        chpStick := make(chan ChpStick, 1)
        chpStick <- ChpStick{}
        t.chpStickChs = append(t.chpStickChs, chpStick)
    }
}
```



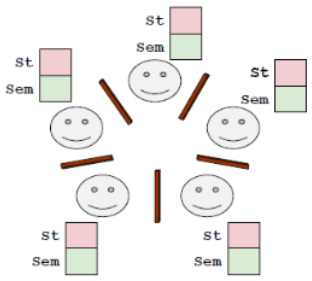
```go
func (t *DiningTable2) Eat(pid int, eat_callback func(pid int)) {
    evenChpStickCh := t.evenChpStickCh(pid)
    oddChpStickCh := t.oddChpStickCh(pid)

breakLock:
    for {
        <-evenChpStickCh
        select {
        case <-oddChpStickCh:
            break breakLock
        default:
            // Couldn't get oddChpStickCh, swap order of chpSticks
        }
        evenChpStickCh <- ChpStick{}

        <-oddChpStickCh
        select {
        case <-evenChpStickCh:
            break breakLock
        default:
            // Couldn't get evenChpStickCh, swap order of chpSticks
        }
        oddChpStickCh <- ChpStick{}
    }

    eat_callback(pid)

    evenChpStickCh <- ChpStick{}
    oddChpStickCh <- ChpStick{}
}
```

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

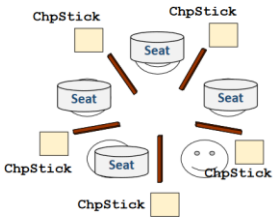
```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

        state[ i ] = EATING;
        signal( s[i] );
    }
}
```




```
void putChpStcks( i )
{
        wait( mutex );

        state[i] = THINKING;
        safeToEat( LEFT );
        safeToEat( RIGHT );

        signal( mutex );
}
```

Dining philosophers: limited eater

```
void philosopher(int i){

    while (TRUE){
        Think( );
        wait(seats);
        wait(chpStick[LEFT]);
        wait(chpStick[RIGHT]);
        Eat( );
        signal(chpStick[LEFT]);
        signal(chpStick[RIGHT]);
        signal(seats);
    }

}
```



Seats = chpsticks – 1

- No deadlock

C++: Dining philosophers' implementation (2)

- `footman_sem` used to limit the number of eaters
- Starvation is possible because mutexes and semaphores are not fair in C++

```
30  template <size_t NumP>
31  struct DiningTable2 {
32    using ChpStick = std::mutex;
33
34    ChpStick chpSticks[NumP];
35    std::counting_semaphore<> footman_sem{NumP - 1};
36
37    // pid = philosopher id
38    ChpStick& get_left_chpStick(size_t pid) { return chpSticks[pid];
39    ChpStick& get_right_chpStick(size_t pid) { return chpSticks[(pid
40
41    void eat(size_t pid, void (*eat_callback)(size_t pid)) {
42      footman_sem.acquire();
43      std::scoped_lock left_lock{get_left_chpStick(pid)};
44      std::scoped_lock right_lock{get_right_chpStick(pid)};
45      eat_callback(pid);
46      footman_sem.release();
47    }
48  };
```


C++: Dining philosophers' implementation (3)

- Use a fair semaphore
  - FIFO semaphore will be discussed in tutorial 7

```
69  template <size_t NumP>
70  struct DiningTable3 {
71    using ChpStick = FairMutex;
72
73    ChpStick chpSticks[NumP];
74    FairSemaphore footman_sem{NumP - 1};
75
76    // pid = philosopher id
77    ChpStick& get_left_chpStick(size_t pid) { return chpSticks[p
78    ChpStick& get_right_chpStick(size_t pid) { return chpSticks[
79
80    void eat(size_t pid, void (*eat_callback)(size_t pid)) {
81      footman_sem.acquire();
82      std::scoped_lock left_lock{get_left_chpStick(pid)};
83      std::scoped_lock right_lock{get_right_chpStick(pid)};
84      eat_callback(pid);
85      footman_sem.release();
86    }
87  };
```

- This time it doesn't matter what order each philosopher tries to obtain each chopsticks

```
126 type DiningTable3 struct {
127     numPhilosophers int
128     chpStickChs         []chan ChpStick
129     footman         chan struct{}
130 }
131
132 func (t *DiningTable3) Init(numPhilosophers int) {
133     t.numPhilosophers = numPhilosophers
134
135     t.chpStickChs = make([]chan ChpStick, 0, numPhilosophers)
136     for i := 0; i < numPhilosophers; i++ {
137         chpStick := make(chan ChpStick, 1)
138         chpStick <- ChpStick{}
139         t.chpStickChs = append(t.chpStickChs, chpStick)
140     }
141
142     t.footman = make(chan struct{}, numPhilosophers-1)
143     for i := 0; i < numPhilosophers-1; i++ {
144         t.footman <- struct{}{}
145     }
146 }
```

```
155 func (t *DiningTable3) Eat(pid int,
156         eat_callback func(pid int)) {
157     leftChpStickCh := t.leftChpStickCh(pid)
158     rightChpStickCh := t.rightChpStickCh(pid)
159
160     <-t.footman
161
162     select {
163     case <-leftChpStickCh:
164         <-rightChpStickCh
165     case <-rightChpStickCh:
166         <-leftChpStickCh
167     }
168
169     eat_callback(pid)
170
171     leftChpStickCh <- ChpStick{}
172     rightChpStickCh <- ChpStick{}
173     t.footman <- struct{}{}
174 }
```

Problem description

- 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.

	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)`

C++: Barbershop implementation

```
23  void customer(void (*balk)(), void (*getHairCut)()) {
24    {
25      std::scoped_lock lock{mut};
26      if (customers == MaxCustomers + 1) {
27        balk();
28        return;
29      }
30      customers++;
31    }
32
33    customer_sem.release();
34    barber_sem.acquire();
35    getHairCut();
36    done_customer_sem.release();
37    done_barber_sem.acquire();
38
39    {
40      std::scoped_lock lock{mut};
41      customers--;
42    }
43  }
44  };
```

```
4   template <size_t MaxCustomers>
5   struct Barbershop1 {
6     size_t customers;
7     std::mutex mut;
8     std::counting_semaphore<> customer_sem;
9     std::counting_semaphore<> barber_sem;
10    std::counting_semaphore<> done_customer_sem;
11    std::counting_semaphore<> done_barber_sem;
12
13    void barber(void (*cutHair)()) {
14      while (true) {
15        customer_sem.acquire();
16        barber_sem.release();
17        cutHair();
18        done_customer_sem.acquire();
19        done_barber_sem.release();
20      }
21    }
```

Go: Barbershop implementation

- Using channels
  - Customers might not get served in the FIFO order

```
23 func (bs *Barbershop1) Customer(balk func(), getHairCut func()) {
24     select {
25     case bs.chairs <- 1:
26         // Sat in chair, ask for barber
27         <-bs.barberAvailable
28         // Get hair cut
29         getHairCut()
30     default:
31         // Chairs are full
32         balk()
33     }
34 }
```

```
3  type Barbershop1 struct {
4      chairs          chan int
5      barberAvailable chan struct{}
6  }
7
8  func (bs *Barbershop1) Init(numChairs int) {
9      bs.chairs = make(chan int, numChairs)
10     bs.barberAvailable = make(chan struct{})
11 }
12
13 func (bs *Barbershop1) Barber(cutHair func()) {
14     for {
15         bs.barberAvailable <- struct{}{}
16         // Someone is sitting in the chair, free them
17         <-bs.chairs
18         // Cut their hair
19         cutHair()
20     }
21 }
```

Variations

- FIFO barbershop
- Multiple barbers

Summary

- Take advantage of the specification of the programming language when possible
- No fair semaphore
- Deadlock avoidance in
  - Acquiring multiple locks in C++
  - Odd-even ring communication

Tutorial 7: Classic concurrency problems in C++ and Go

H2O Problem

The task is to implement a "water factory". Hydrogen and oxygen atoms arrive at the factory, and when 2 hydrogen and 1 oxygen atom arrive, the three atoms should bond together.

However, to prevent illegal bonding, only the 3 atoms involved should be concurrently executing the bond, and all other atoms should be blocked until the bonding process is complete. Furthermore, atoms should not start bonding until 3 atoms are ready to bond.

For example, if 4 hydrogen and 2 oxygen atoms arrive, 2 of the hydrogen atoms and 1 of the oxygen atoms should unblock and call bond, and only after all 3 atoms have finished calling bond should the other 3 atoms unblock and call bond.

C++ Interface

```
struct WaterFactory0 {
  void oxygen(void (*bond)()) { bond(); }
  void hydrogen(void (*bond)()) { bond(); }
};
```

Go Interface

```
type WaterFactoryInterface interface {
          hydrogen(func())
          oxygen(func())
}
```

Demo 1: (C++) Using a barrier

```
struct WaterFactory1 {
  std::barrier<> barrier;

  WaterFactory1() : barrier{3} {}

  void oxygen(void (*bond)()) {
    barrier.arrive_and_wait();
    bond();
  }

  void hydrogen(void (*bond)()) {
    barrier.arrive_and_wait();
    bond();
  }
};
```

INCORRECT usage of barrier: we unfortunately don't enforce that they're of the correct types.

```
auto oxygen_bond = []() {
  print("Bonding oxygen\n");
  std::this_thread::sleep_for(std::chrono::milliseconds{5});
  print("Done\n");
};
std::thread t1{[&]() { factory.oxygen(oxygen_bond); }};
std::thread t2{[&]() { factory.oxygen(oxygen_bond); }};
std::thread t3{[&]() { factory.oxygen(oxygen_bond); }};
// Bonding oxygen
// Bonding oxygen
// Bonding oxygen
// Done
// Done
// Done
```

To ensure that only the correct atoms go through, we can add semaphores around the barrier to limit the number of atoms of each type in the critical section. This is essentially a slightly weaker form of mutual exclusion.

```
  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
  }
```

Aside: std::barrier vs std::latch
Question: What if we use an std::latch instead of an std::barrier?
A: Unlike std::latch, std::barrier are reusable.

1.2 Demo 2: (Go) Using a daemon goroutine

In C++, we modelled the problem as groups of 3 atoms passing a barrier. In Go, we can interpret waiting for 3 atoms as a looping process, and use communication to achieve the desired outcome. Our first solution in Go is to use a daemon goroutine to help coordinate all the atoms.

The daemon is fully in charge of everything that happens. No bonding starts unless the daemon says so, and it only starts processing the next water molecule when it knows the current molecule is fully bonded.

The daemon will run the following algorithm:
- (Precommit) Receive arrival requests from 2 hydrogen atoms and 1 oxygen atom
- (Commit) Tell all 3 atoms to proceed and begin bonding
- (Postcommit) Wait for all 3 atoms to finish bonding and tell us to continue
- Go to step 1

(As an aside: this uses language similar to "Two-Phase Commit" protocols (e.g., here))

In order to communicate directly with the atom goroutines in steps 2 and 3, we will use a chan struct{}. We can share this communication channel by sending it over a higher order channel in step 1.

An atom will run the following algorithm:
- Create private communication channel
- (Precommit) Send channel in an arrival request to the daemon
- (Commit) Receive the signal to proceed from the daemon
- Bond
- (Postcommit) Send the signal that we've bonded to the daemon

```
type WaterFactoryWithDaemon struct {
  // Channels for atoms to send their arrival requests
  precomH chan chan struct{}
  precomO chan chan struct{}
}

func NewFactoryWithDaemon() WaterFactoryWithDaemon {
  wfd := WaterFactoryWithDaemon{
    precomH: make(chan chan struct{}),
    precomO: make(chan chan struct{}),
  }

  // Daemon goroutine
  go func() {
    for {
      // Step 1: (Precommit)
      //         Receive arrival requets from 2 hydrogen and 1 oxygen
atoms
      h1 := <-wfd.precomH
      h2 := <-wfd.precomH
      o := <-wfd.precomO

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

      // Step 3: (Postcommit)
      //         Wait until the 3 atoms have finished before looping
      // We re-use the same communication channel as (Commit)
      <-h1
      <-h2
      <-o
    }
  }()

  return wfd
}

func (wfd *WaterFactoryWithDaemon) hydrogen(bond func()) {
  commit := make(chan struct{}) // Step 1: Create private communication
channel
  wfd.precomH <- commit         // Step 2: (Precommit)
  <-commit                      // Step 3: (Commit)
  bond()                        // Step 4: Bond
  commit <- struct{}{}          // Step 5: (Postcommit)
}

func (wfd *WaterFactoryWithDaemon) oxygen(bond func()) {
  commit := make(chan struct{}) // Step 1: Create private communication
channel
  wfd.precomO <- commit         // Step 2: (Precommit)
  <-commit                      // Step 3: (Commit)
  bond()                        // Step 4: Bond
  commit <- struct{}{}          // Step 5: (Postcommit)
}
```

1.2.1 Aside: Downsides of a daemon based approach

However, using a daemon has some downsides. Since the daemon runs on a goroutine and has references to the water factory object, even if all other references to the water factory are dropped, the water factory will never be cleaned up as the daemon is still holding a reference to it. The daemon goroutine itself also does not go away and leaks memory.

We can somewhat remedy this by giving our water factory a Destroy method that shuts down the daemon with a Context, but this forces the user to keep track of references to the water factory, and to properly clean it up when the last reference is about to be dropped. This means that such a water factory effectively needs manual memory management, despite Go being a garbage collected language!

Separately: the daemon is a bottleneck and a single point of failure in this design and we can implement something better.

1.3 Demo 3: (Go) Using oxygen atoms as leader goroutines

In this case, it is easy to avoid creating a daemon goroutine by electing a "leader" that the other atoms will communicate with. Since there is only one oxygen atom in a water molecule, we can make oxygen atoms leaders.

To ensure that there's never two leaders active at the same time, we can simply use a mutex.

Hydrogen atoms will run the following algorithm, as before:
- Create private communication channel
- (Precommit) Send channel in an arrival request to the daemon
- (Commit) Receive the signal to proceed from the daemon
- Bond
- (Postcommit) Send the signal that we've bonded to the daemon

Oxygen atoms will combine a single iteration of the daemon and the algorithm for atoms:
- Become leader by locking mutex
- (Precommit) Receive arrival requests from 2 hydrogen atoms
- (Commit) Tell the hydrogen atoms to proceed and begin bonding
- Bond
- (Postcommit) Wait for the hydrogen atoms to finish bonding
- Step down from being leader by unlocking mutex

```
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
}

func (wf *WaterFactoryWithLeader) hydrogen(bond func()) {
  commit := make(chan struct{}) // Step 1: Create private communication
channel
  wf.precomH <- commit          // Step 2: (Precommit)
  <-commit                      // Step 3: (Commit)
  bond()                        // Step 4: Bond
  commit <- struct{}{}          // Step 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 requets 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{}{}
}
```

C++ Implementations

```
// (Incorrect)
// Let's try to use a barrier to only allow 3 atoms into the bonding
// section at a time.
//
// Unfortunately, this doesn't enforce that the 3 atoms that get to bond
// are of the right type.
struct WaterFactory1 {
  std::barrier<> barrier;

  WaterFactory1() : barrier{3} {}

  void oxygen(void (*bond)()) {
    barrier.arrive_and_wait();
    bond();
  }

  void hydrogen(void (*bond)()) {
    barrier.arrive_and_wait();
    bond();
  }
};

////////////////////////////////////////////////////////////////

// (Incorrect)
// To solve the type problem, let's try using a semaphore to only let
// atoms of the right type into the barrier.
//
// Now, atoms will cross the barrier in groups of 3, and they're of the
// right type.
//
// However, because bond is simply run concurrently, we don't stop
illegal
// bonding (too many atoms can cross the barrier before they all run
bond)
struct WaterFactory2 {
  std::counting_semaphore<> oxygenSem;
  std::counting_semaphore<> hydrogenSem;
  std::barrier<> barrier;

  // Initialise atom sems to only let 1 oxygen and 2 hydrogen through
  WaterFactory2() : oxygenSem{1}, hydrogenSem{2}, barrier{3} {}

  void oxygen(void (*bond)()) {
    oxygenSem.acquire();  // Lets at most one oxygen through
    barrier.arrive_and_wait();
    oxygenSem.release();  // When 3 atoms have passed the barrier, we let
                          // the next atom in
    bond();
  }

  void hydrogen(void (*bond)()) {
    hydrogenSem.acquire();  // Lets at most 2 hydrogens through
    barrier.arrive_and_wait();
    hydrogenSem.release();  // When 3 atoms have passed the barrier, we
                            // let the next atom in
    bond();
  }
};

////////////////////////////////////////////////////////////////

// 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
  }
};

////////////////////////////////////////////////////////////////

// Alternative solution
// Blatantly ripped off from the Little Book of Semaphores
struct WaterFactory4 {
  std::counting_semaphore<> mut;

  // Holds the 3 atoms back until they've all bonded
  std::barrier<> barrier;

  // Number of oxygen and hydrogen waiters
  uint64_t numOxygen;
  uint64_t numHydrogen;

  // Blocks a set of oxygen and hydrogen waiters
  std::counting_semaphore<> oxygenQueue;
  std::counting_semaphore<> hydrogenQueue;

  WaterFactory4()
      : mut{1},
        barrier{3},
        numOxygen{0},
        numHydrogen{0},
        oxygenQueue{0},
        hydrogenQueue{0} {}

  void oxygen(void (*bond)()) {
    mut.acquire();

    // Advertise ourselves as an oxygen waiter
    numOxygen += 1;

    if (numHydrogen >= 2) {
      hydrogenQueue.release(2);
      numHydrogen -= 2;
      oxygenQueue.release();
      numOxygen -= 1;
    } else {
      mut.release();
    }

    oxygenQueue.acquire();
    bond();

    barrier.arrive_and_wait();
    mut.release();
  }

  void hydrogen(void (*bond)()) {
    mut.acquire();
    numHydrogen += 1;
    if (numHydrogen >= 2 && numOxygen >= 1) {
      hydrogenQueue.release(2);
      numHydrogen -= 2;
      oxygenQueue.release();
      numOxygen -= 1;
    } else {
      mut.release();
    }

    hydrogenQueue.acquire();
    bond();

    barrier.arrive_and_wait();
  }
};
```

2 FIFO Semaphore

A semaphore is a counter that can be incremented (release) and decremented (acquire), but blocks when one attempts to decrement the counter past 0.

Threads (or goroutines) that are blocked at a semaphore are unblocked when another thread (or goroutine) increments the counter above 0, so that the decrementing thread can proceed. We say that in this case, the release wakes up one of the waiters.

However, if there are multiple threads blocked at a semaphore, there is no guarantee which waiter is woken up. This sometimes leads to resource starvation, where a thread is blocked at a semaphore for an unfairly long time.

For example, the footman solution to the Dining Philosophers problem is starvation free but only if the semaphore and mutexes used are also starvation free. If the platform does not provide us with such guarantees, using one of the implementations shown below may give us starvation freedom even if the builtin primitives are not.

t1: sleep(1); sem.acquire(); print("t1 woke up");
t2: sleep(2); sem.acquire(); print("t2 woke up");
t3: sleep(3); sem.release();
Either `t1 woke up` or `t2 woke up` can be printed even if t1 acquires the semaphore first!

t1: sleep(1); fifo_sem.acquire(); print("t1 woke up");
t2: sleep(2); fifo_sem.acquire(); print("t2 woke up");
t3: sleep(3); fifo_sem.release();
`t1 woke up` must be printed if t1 reaches the fifo_sem first (highly probably in this example).

We can think of FIFO semaphores as having two parts: a counter and a queue of waiters.

```
struct FIFOSemaphore0 {
  std::ptrdiff_t count;

  FIFOSemaphore0(std::ptrdiff_t initial_count) : count{initial_count} {}
  void acquire() { count--; }
  void release() { count++; }
};

type SemaphoreInterface interface {
          Acquire()
          Release()
}
```

Demo 4: (C++) Using a ticket queue

We can implement a queue of waiters by using a ticket queue (e.g. like in a clinic). Arrivals obtain a queue number from the receptionist, and when they're ready to be released, the "now serving" number is incremented to their queue number or higher.

We can implement this with atomics in C++:

```
struct FIFOSemaphore2 {
  // The next ticket we will hand out
  alignas(128) std::atomic<std::ptrdiff_t> next_ticket;
  // The "now serving" number
  alignas(128) std::atomic<std::ptrdiff_t> now_serving;

  // Initialise with `initial_count` requests "pre-served"
  FIFOSemaphore2(std::ptrdiff_t initial_count)
      : next_ticket{1}, now_serving{initial_count} {}

  void acquire() {
    // When a thread arrives, get a ticket
    std::ptrdiff_t my_ticket =
        next_ticket.fetch_add(1, std::memory_order_relaxed);
    // Wait until the now serving number reaches or passes us
    while (now_serving.load(std::memory_order_acquire) < my_ticket) {
    }
  }

  void release() {
    // Increment the now serving number
    now_serving.fetch_add(1, std::memory_order_acq_rel);
  }
};
```

Coincidentally, this also solves the problem of keeping track of the semaphore count. A positive count corresponds to the case where the "now serving" number is larger than the last ticket we handed out. Thus, if a new thread acquires the semaphore, the ticket it gets will already be less than or equal to the "now serving" number, and it can proceed without blocking.

Task 1: Replacing the spinwait with a condition variable

We can improve the design for workloads with high contention by not spinning needlessly when the semaphore is blocked. Instead, we can replace the spinwait with a condition variable (or if you're adventurous, std::atomic<T>::wait works too!)

```
  std::condition_variable cond;
  std::mutex mut;
  ...
  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; });
  }
```

Demo 5: (C++) Using a queue of semaphores

Another solution is to somehow be able to pick which waiter to wake up, and simply wake them up in FIFO order. The easiest way to do this is to block each waiter on a separate semaphore; these semaphores can be tracked in a queue.

In order for releases to unblock acquires, each waiter adds a waiter-specific semaphore to the queue and then blocks on it. Releases can unblock waiters by popping them from the queue and releasing the waiter-specific semaphore.

```
struct FIFOSemaphore5 {
  struct Waiter {
    std::binary_semaphore sem{0};
  };

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

  FIFOSemaphore5(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
  }
};
```

Aside: Intrusive linked list

The above solution is not ideal because of the large number of allocations required (roughly 2 allocations every time a new waiter is added to the queue). We can mitigate this by using an intrusive linked list.

With an intrusive linked list, we allocate the linked list nodes on the stack rather than in heap memory, and as such the "linked list" structure is not a standalone object but rather is embedded as part of a larger one (in this case, the stack).

Since stack allocation is essentially free and very cache friendly, this has the potential to speed up the semaphore by a decent factor. You may peruse an implementation of FIFO semaphore using an intrusive linked list of semaphores in the full Godbolt link | full Fsmbolt link, under FIFOSemaphore7.

```
// Linked list of semaphore: Use intrusive linked list to reduce
allocations
struct FIFOSemaphore7 {
  struct Waiter {
    std::binary_semaphore sem{0};
    Waiter *next{nullptr};
  };

  std::mutex mut;
  Waiter *front;
  Waiter *back;
  std::ptrdiff_t count;

  FIFOSemaphore7(std::ptrdiff_t initial_count)
      : mut{}, front{nullptr}, back{nullptr}, count{initial_count} {}

  void acquire() {
    Waiter waiter;

    {
      std::scoped_lock lock{mut};

      // Optimistically check if we can proceed
      if (count > 0) {
        count--;
        return;
      }

      // Sadly we are blocked

      // First append ourselves to the waiter list
      if (back == nullptr) {
        front = back = &waiter;
      } else {
        back->next = &waiter;
        back = &waiter;
      }
    }

    waiter.sem.acquire();
  }

  void release() {
    Waiter *head;
    {
      std::scoped_lock lock{mut};
      if (front == nullptr) {
        count++;
        return;
      }
      head = front;
      front = front->next;
      if (front == nullptr) {
        back = nullptr;
      }
    }
    head->sem.release();
  }
};
```

Demo 6: (Go) Using a buffered channel

In Go, we can use buffered channels as semaphores. Rather than using the channel to send values, instead the number of empty slots in the channel acts as the counter for a semaphore.

Sends to such a channel decrement the number of empty slots, and receives increment the number of empty slots.

When the number of elements in the channel reaches the capacity of the channel, the number of remaining slots drops to 0, and this is also precisely when sends will begin to block.

```
type Semaphore1 struct {
          sem chan struct{}
}

func NewSemaphore1(capacity int, initial_count int) *Semaphore1 {
          sem := Semaphore1{
                    sem: make(chan struct{}, capacity),
          }
          for ; initial_count < capacity; initial_count++ {
                    sem.Acquire()
          }
          return &sem
}
```

```
func (s *Semaphore1) Acquire() {
          // Send to the channel to decrement the number of empty slots
          // Blocks if there are no slots remaining
          s.sem <- struct{}{}
          // Blocked goroutines will be unblocked in FIFO order as of Go
             1.17
}

func (s *Semaphore1) Release() {
          // Receive from the channel to increment the number of empty
             slots
          <-s.sem
}
```

While this is indeed a semaphore, is it FIFO?

In practice, the answer is yes (as of Go 1.17). Due to the way channels are implemented in the Go runtime, waiters are indeed unblocked in the order they begin blocking.

However, as the **Go specification does not guarantee that waiters are unblocked in FIFO order**, this behaviour may change in a future version of Go.

Demo 7: (Go) Using a daemon goroutine

Now, let's not assume that channels unblock waiters in FIFO order. Once again, we can use a daemon goroutine to coordinate the others. This is similar to the C++ solution with a queue of semaphores, except that one goroutine is responsible for managing the queue, rather than it being a shared data structure.

Releasers and acquirers communicate with the daemon goroutine over a couple channels. Whenever the daemon receives a request for acquire, it either decrements the counter or adds a new waiter to a queue. Whenever the daemon receives a request for release, it either increments the counter or wakes up the oldest waiter in the queue.

To allow the daemon to choose when acquirers should unblock, each acquire sends a channel to the daemon and then blocks on it by trying to receive from it. The daemon can then unblock the acquire by sending to it.

```
type Semaphore2 struct {
    acquireCh chan chan struct{}
    releaseCh chan struct{}
}

func NewSemaphore2(initial_count int) *Semaphore2 {
    sem := new(Semaphore2)
    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
waiters
        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 *Semaphore2) 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 *Semaphore2) Release() {
    s.releaseCh <- struct{}{}
}

/////

func (s *Semaphore2) 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
}
```

Is this really FIFO?

Technically, it is possible that an acquire is blocked on the first send to s.acquireCh, even before it's able to send its channel to the daemon. If we assume that channels do not unblock in FIFO order, it's possible that it remains blocked on this first send forever while other goroutines are constantly sending new acquire requests to the daemon.

So the answer is no, it's not actually FIFO in the sense that a goroutine A that calls Acquire can be blocked before another goroutine B, and yet goroutine B unblocks before goroutine A.

However, the ordering is enforced from the moment that the first send actually succeeds. Since the daemon is capable of emptying its request queues relatively quickly, and the request queues can be buffered to a length where it does not block in practice, it is possible to make an argument that this semaphore is FIFO under certain conditions.

L9 - Safety in Concurrent Programming with Rust

Challenges in concurrent programming

- Parallelizing ... anything is a daunting task
  - The goal is to make things faster
  - Many times, parallelizing is done by just adding another instance that does the same work
- Race conditions, data races, deadlocks, starvation
- Unsafe usage of memory in C/C++
  - Use after free (UAF)
  - Double free

Fearless Concurrency in Rust

Rust was initiated with two goals in mind:

- Safety in system programming
- Painless concurrency

Rust nowadays

- Strong safety guarantees
  - No seg-faults, no data races, expressive type system
- Without compromising on performance
  - No garbage collector, no runtime
  - Same level of performance as C/C++
- Goal
  - **Confident, productive systems programming**
- Compiler can optimize code because of the safety guarantees that rust has (no memory bug in program)

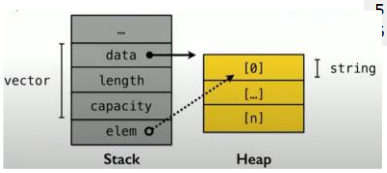
Rust

- Rustup – to install your rust tools
- Rustc – the Rust compiler
- Cargo
  - Calls the compiler – rustc
  - TOML (Tom's Obvious, Minimal Language) format for the configuration file
- Packages, crates, modules
  - A package is one or more crates that provide a set of functionality
  - A crate is a binary or library
  - Modules are used to organize code within a crate into groups
    - Privacy control

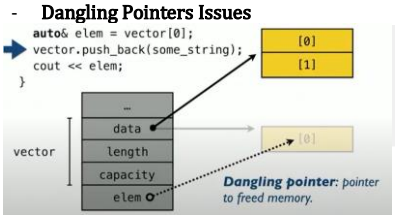
C++ is unsafe

- Vector is freed when we exit the scope

```
1  void example() {
2      vector<string> vector;
3      ...
4      auto& elem = vector[0];
5      ...
6  }
```

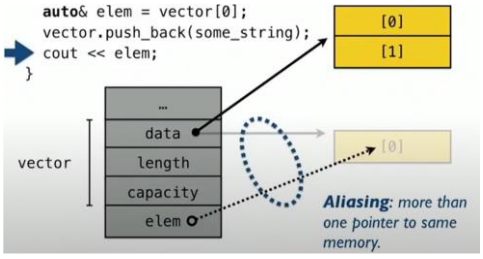


- **Dangling Pointers Issues**




```
void example() {
    vector<string> vector;
    ...
    auto& elem = vector[0];
    vector.push_back(some_string);
    cout << elem;
}
```

- Aliased pointers –pointers that point to the same chunk of memory
  - elem and vector[0]
- Mutation –changing a pointer
- **Aliasing + mutation** –changing (modifying) pointers that point to the same chunk of memory



Solution

- **Ownership and borrowing**
  - Prevent simultaneous mutation and aliasing
- No runtime like in C++
- Memory safety like in garbage collected languages
- **No data races** like in …Rust

Ownership (1)

- Lines 2-4: Vector bookis initialized, Owner: main function
- Line 5: give ownership to publish (pass the without &)

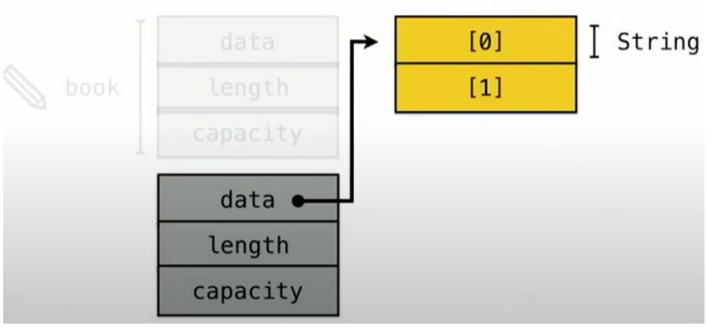
Runtime:

- Copy over the fields from main's stack to publish's stack
- Forget about the first book in main

Line 11: Runs the destructor for book

Line 8: Compilation error: use of moved value book.
Ownership does not allow aliasing!

```
1  fn main() {
2      let mut book = Vec::new();
3      book.push(...);
4      book.push(...);
5      publish(book);
6      // a second call to publish would
7      // generate a compilation error
8      // publish(book);
9  }
10 fn publish(book: Vec<String>) {
11     ...
12 }
```



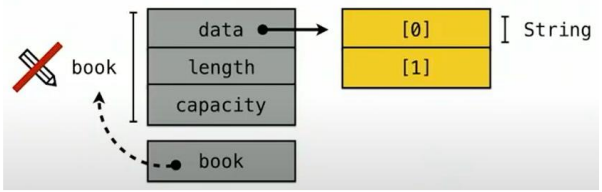
Rust ownership compared to C++

- Rust: giving ownership is the default
  - Not like the copy constructor in C++
  - A bit like a move in C++, but **enforced at compilation time and no ownership is retained**
- Rust: deep copy of data is explicit using clone()
  - In C++, the copy constructor does a deep copy

Shared Borrow

Line 12: type is a reference to a vector -> use &
Line 5, 10: **borrow** the vector, creating a **shared reference.**
A shared borrow allows aliasing, but no mutation!

```
1  fn main() {
2      let mut book = Vec::new();
3      book.push(...);
4      book.push(...);
5      publish(&book);
6      // a second call to publish
7      // borrows again the reference
8      // to book.
9      // compilation is successful
10     publish(&book);
11 }
12 fn publish(book: &Vec<String>) {
13     ...
14 }
```



Consequences of shared borrow

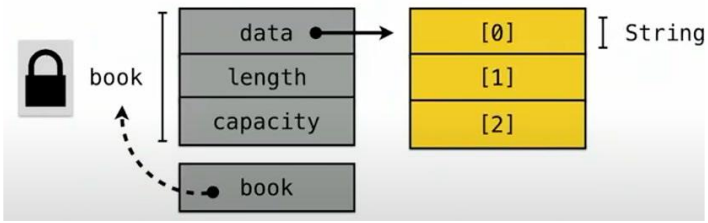
```
1  fn example() {
2      let mut vector = Vec::new();
3      ...
4      let elem = &vector[0];
5
6      // mutation is not allowed while
7      // a shared borrow exists for book.
8      // compilation error
9      vector.push(some_string);
10     ...
11 }
```

Line 4: vector is (shared) borrowed here
- Freezes the **whole container vector**

Line 9: cannot mutate (compilation error)

Mutable Borrow

```
1  fn main() {
2      let mut book = Vec::new();
3      book.push(...);
4      book.push(...);
5      publish(&mut book);
6      publish(&mut book);
7  }
8
9  fn publish(book: &mut Vec<String>) {
10     book.push(...);
11 }
```



Line 9: mutable reference to a vector
Line 5,6: mutable borrow

Can have exactly one mutable ref at a single time

```
1  let mut book = Vec::new();
2  book.push(...); //success: book is mutable
3  {
4      let r = &book; // shared borrow of book
5      book.push();   // compilation error: cannot mutate
6                     //   while shared
7      r.push(...);   // compilation error: cannot mutate
8                     //   while shared
9  }   // shared borrow ends
10 book.push(...);    // success: book can be mutated
```

```
12 let mut book = Vec::new();
13 book.push(...);        //success: book is mutable
14 {
15     let r = &mut book; // mutable borrow of book
16     book.len();        // compilation error: cannot access
17                        //   while mutable borrow exists
18     r.push(...);       // success: reference can be mutated
19 }   // mutable borrow ends
20 book.push(...);        // success: book can be mutated
```

Memory safety in Rust
- The borrow checker statically prevents aliasing + mutation
  - Compile time
  - Fighting the borrow checker! Don't give up! Don't use unsafe!
- **Ownership** prevents **double-free**
  - The owner frees
- **Borrowing** prevents **use-after-free**
- No segfaults!

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

No data races in Rust
- Data race = sharing + mutation + no ordering
- Sharing + mutation are prevented in Rust

Library-based concurrency
- Not build into the language
  - Rust had message passing build into the language –removed
- Library-based –in std or other libraries
  - Multi-paradigm
  - Leverage on ownership/borrowing

Create a thread:
Line 1: create a thread
- loc is a JoinHandle
- If loc is dropped, the spawned thread is detached

Line 5: join a thread

```
1 let loc = thread::spawn(|| {
2     "world"
3 });
4 println!("Hello, {}!",
5     loc.join().unwrap());
```

Transfer the vector to a thread
- **Move** converts any variables captured by reference or mutable reference to variables captured by value
- Move keyword: the **closure will take ownership of the values it uses** from the environment, thus transferring ownership of those values from one thread to another
- Line 5: **error:** use after move

```
1 let mut dst = Vec:: new();
2 thread::spawn(move || {
3     dst.push(3);
4 });
5 dst.push(3);
```

Remove the move
Line 3: error: value doesn't live long enough
- Possible memory issues: UAF
- Compilation Error

```
1 let mut dst = Vec:: new();
2 thread::spawn(|| {
3     dst.push(3);
4 });
5 dst.push(3);
```

Spawn a thread
- Capture everything as a borrow
- The close captures dst(mutableborrow)
- Rust infers how to capture dst

Reference Counting (RC)
Line 4: error: Rc<T> can't be sent across threads.
- RC type is not atomically managed
- No send trait

```
1 let v = Rc:: new(vec![1 ,2, 3]);
2 let v2 = v.clone();
3 thread::spawn(move || {
4     println!("{}", v.len());
5 });
6 another_fn(v2.clone);
```

Traits
- Like an interface that you can implement for a given type
- It might have methods

Example
- Clone

Marker traits:
- **Send** – transferred across thread boundaries
  - String, u32, Arc<String>
- **Sync** –safe to share references between threads
  - Type T is Sync if and only if &T is Send
- **Copy** –safe to memcpy(for built-in types)
  - u32, f32
  - Not for Strings

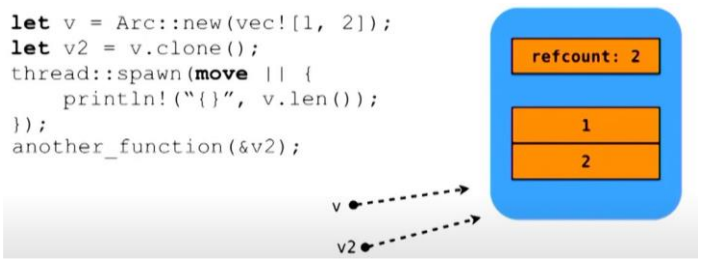
```
trait Clone {
  fn clone(&self) -> Self;
}
```

```
impl<T: Clone> Clone for Vec<T> {
  fn clone(&self) -> Vec<T> {
    let mut v = Vec::new();
    for elem in self {
        v.push(elem.clone());
    }
    return v;
  }
}
```

Atomically Reference Counted -Arc
- Arc: allows only shared references
  - References cannot be mutated

Line 3: move reference
Line 4: it's safe to access v

```
let v = Arc::new(vec![1, 2]);
let v2 = v.clone();
thread::spawn(move || {
    println!("{}", v.len());
});
another_function(&v2);
```



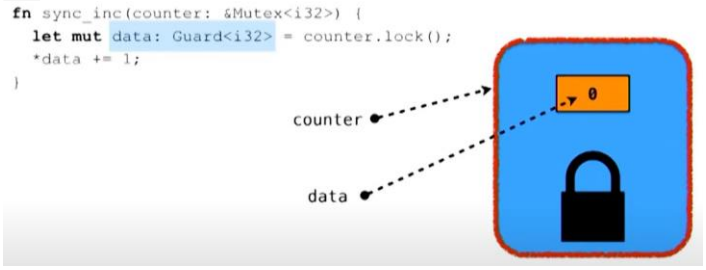
Synchronization in Rust

- Shared memory
  - Mutex
  - Atomics
- Message-passing
  - Channels: MPSC (multi-producer, single-consumer FIFO queue)

Mutex

- Based on ownership
- Data protected by the mutex
- Lock returns a guard - a proxy through which we can access the data

```
1 fn sync_inc(counter: &Mutex<i32>) {
2     let mut data Guard<i32> = counter.lock();
3     *data += 1;
4 }
```



Atomics

- Similar to modern C++
  - Same memory model
- Ordering of memory operations -SeqCst

```
1 let number = AtomicUsize::new(10);
2 let prev = number.fetch_add(1, SeqCst);
3 assert_eq!(prev, 10);
4 let prev = number.swap(2, SeqCst);
5 assert_eq!(prev, 11);
6 assert_eq!(number.load(SeqCst), 2);
```

L1: number is not mutable. Mutate through a shared ref.

Multi-Producer, Single-Consumer FIFO queue

Channel with a reading and writing reference

- Accepts one reader and multiple writers

```
1 let (tx, rx) = mpsc::channel ();
2 let tx2 = tx.clone();
3 thread::spawn(move || tx.send(5));
4 thread::spawn(move || tx2.send(4));
5
6 //prints 4 and 5 in an unspecified order
7 println!("{:?}", rx.recv));
8 println!("{:?}", rx.recv));
```

// If use tx and tx2 in one thread -> error
// Single consumer: can't clone reading end of the channel

Crossbeam

- Ability to create scoped threads –now in std
  - Scope is like a little container we are going to put our threads in
  - You cannot borrow variables mutably into two threads in the same scope
- Message passing using multiple-producer-multiple-consumer channel
  - With exponential backoff

```
1  fn main() {
2      let v = vec![1, 2, 3];
3      println!("main thread has id {}", thread_id::get());
4
5      std::thread::scope(|scope| {
6          scope.spawn(|inner_scope| {
7              println!("Here's a vector: {:?}", v);
8              println!("Now in thread with id {}", thread_id::get());
9          });
10     }).unwrap();
11
12     println!("Vector v is back: {:?}", v);
13 }
```

L5: Create the scope
L6: Spawn threads

Producer Consumer

```
11 fn main() {
12     let (send_end, receive_end) = bounded(CHANNEL_CAPACITY);
13
14     let mut threads = vec![];
15     for _i in 0 .. NUM_THREADS {
16         let send_end = send_end.clone();
17         threads.push(
18             thread::spawn(move || {
19                 for _k in 0 .. ITEMS_PER_THREAD {
20                     let produced_value = produce_item();
21                     send_end.send(produced_value).unwrap();
22                 }
23             })
24         );
25     }
26
27     for _j in 0 .. NUM_THREADS {
28         // create consumers
29         let receive_end = receive_end.clone();
30         threads.push(
31             thread::spawn(move || {
32                 for _k in 0 .. ITEMS_PER_THREAD {
33                     let to_consume = receive_end.recv().unwrap();
34                     consume_item(to_consume);
35                 }
36             })
37         );
38     }
39
40     for t in threads {
41         let _ = t.join();
42     }
43     println!{"Done!"}
44 }
```

L12: Use a bounded channel

Exponential backoff

- Resources might not be available right now?: Retry later
- It's unhelpful to have a tight loop that simply retries as fast as possible
- Wait a little bit and try again
  - If the error occurs, next time wait a little longer
- Rationale: if the resource is overloaded right now, the reaction of requesting it more will make it even more overloaded and makes the problem worse!

Backoff with scoped threads

Line 12: backoff in lock-free loop
Line 20: wait for another thread to take its turn first

```
1  use crossbeam_utils::Backoff;
2  use std::sync::atomic::AtomicUsize;
3  use std::sync::atomic::Ordering::SeqCst;
4
5  fn fetch_mul(a: &AtomicUsize, b: usize) -> usize {
6      let backoff = Backoff::new();
7      loop {
8          let val = a.load(SeqCst);
9          if a.compare_and_swap(val, val.wrapping_mul(b), SeqCst) == val {
10             return val;
11         }
12         backoff.spin();
13     }
14 }
17 fn spin_wait(ready: &AtomicBool) {
18     let backoff = Backoff::new();
19     while !ready.load(SeqCst) {
20         backoff.snooze();
21     }
22 }
```

Rayon

- A data parallelism library
  - Parallelize some spots without full/major rewrite
- Similar in functionality with OpenMP
  - But OpenMP uses compiler directives
- Rationale: reasonably common that computationally-intensive parts of the program happen in a loop, so parallelizing loops is likely to be quite profitable

Example: Sequential maximum of a vector

```
1  fn main() {
2      let vec = init_vector();
3      let max = MIN;
4      vec.iter().for_each(|n| {
5          if *n > max {
6              max = *n;
7          }
8      });
9
10     println!("Max value in the array is{}",max);
11     if max == MAX {
12         println!("This is the max value for an i64.")
13     }
14 }
```

Example: Maximum of a vector with Rayon

```
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.")
    }
}
```

- Will create # of hreads = # of cores & dynamically allocate work
- Done in compile time & Execution time

Rust in a nutshell

- Zero-cost abstraction (like C/C++)
- **Memory safety & data-race freedom**
- Results in
  - Confident, productive systems programming

Tutorial 8: Safety and Concurrency in Rust

Concurrent Counter

```
use std::thread;
fn main() {
  let mut counter = 0;

  let t0 = thread::spawn(|| { counter += 1; });
  let t1 = thread::spawn(|| { counter += 1; });

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

  println!("{}", counter);
}
```

Issues with direct port

- Counter is being borrowed mutably twice: it is shared and mutable
- Counter does not live long enough

Shared and mutable

This is quite clear: the two threads have both captured a reference to counter, and because they both modify counter, they capture a mutable reference. We have two simultaneous mutable borrows, which is clearly illegal.

Solving this issue

The simplest way to allow mutation to data shared between threads is using Mutex<T>.

1.1.2 Lifetime too short

At first glance, it seems like there should be no issue here: main joins both threads before exiting, so there is no issue here, as counter will not be dropped before both threads terminate.

We can look at the signature of thread::spawn to understand the cause of the error:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T> where
  F: FnOnce() -> T,
  F: Send + 'static,
  T: Send + 'static
```

spawn is a generic function over F and T. There are some trait bounds on F and T. F must implement FnOnce() -> T i.e. it must be a function that returns T.1 Both F and T also must implement the Send trait i.e. it must be safe to send across threads, and they must be 'static.

A token looking like 'x is a lifetime specifier. These are used to specify lifetimes on references and generic bounds. We will not delve further into that in this tutorial. However, 'static is a special lifetime: it is the static lifetime, i.e. the lifetime of a static global variable (which we will look at in section 4), or in other words the lifetime of the entire process. The type of a string literal is &'static str, because string literals are stored in read-only data and therefore exist for the entire duration of the process. //let s: &'static str = "I live forever!";

So what does it mean for F, i.e. the closure, to have a static lifetime? It means that **any data captured by the closure must be owned, or live for at least as long as the static lifetime**. This is actually a limitation of the original std::thread: because (unlike a C++ std::thread) you are not required to join a thread, it is possible that the thread lives "forever", and therefore any data referenced by the thread must stay live for that period. Later in section 2 we look at scoped threads with std::thread::scope, which ensure that threads exit before the spawning function returns, thereby allowing us to spawn threads that capture local variables directly as references.

Solving this issue

For now, we need to make sure that the counter does not get freed before all the threads exit. A common way to share data between threads in Rust is using Arc<T>, equivalent to C++'s std::shared_ptr<T>. Arc<T> will allocate the data on the heap, track the reference count internally, and only free the allocation when there are no more referrers.

Task 1: Fixing the direct port

Let's try to fix the direct port. We simply need to fix the two issues by using the appropriate standard library constructs as mentioned above:

- To solve the shared-and-mutable problem, put the counter in a Mutex.
- To solve the lifetime problem, use an Arc.

We want to share the same data and the mutex across threads, so we should use an Arc<Mutex<i32>> to apply reference counting on the Mutex containing the i32.

Can we simply allow each thread to capture a reference &T to our counter value wrapped in Arc and Mutex?
A: No; we need to make a new Arc instance for each thread. This can be done using Arc::clone, which creates a new Arc instance that points to the same heap allocation, incrementing the reference count. If we were to let each thread capture a reference &T to the Arc instance of the main thread, we would still run into the same lifetime issue.

Lastly, how do we lock the mutex to get access to the contained data?
Answer: Look at Mutex::lock.

Note that this returns a Result, because locking a mutex can fail for reasons we will discuss later. Therefore, you need to handle the Result to get to the MutexGuard which will grant you access to the data. For our purposes, unwrap is sufficient.
MutexGuard is sort of similar to C++'s std::scoped_lock; when the value goes out of scope, the mutex is automatically released.

```
use std::thread;
use std::sync::{Mutex, Arc};
fn main() {
  let counter = Arc::new(Mutex::new(0));

  let t0 = {
    let counter = counter.clone();
    thread::spawn(move || {
      // THE FOLLOWING IS NOT IDIOMATIC RUST
      use std::ops::{Deref, DerefMut};

      let mutex: &Mutex<i32> = counter.deref();
      let lock_result = mutex.lock();
      let mut lock_guard = lock_result.unwrap();
      let counter_ref: &mut i32 = lock_guard.deref_mut();
      *counter_ref += 1;
      // END UNIDIOMATIC RUST
    })
  };
  let t1 = {
    let counter = counter.clone();
    thread::spawn(move || {
      *counter.lock().unwrap() += 1;
    })
  };

  t0.join().unwrap();
  t1.join().unwrap();

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

Notice the lack of a semicolon at the end of the last line of the block. Remember that blocks in Rust are expressions that evaluate to the final expression in the block, so this block will evaluate to the value returned by thread::spawn, which is a JoinHandle.

The lock_guard.deref_mut() is done automatically by our use of the * operator. Dereferencing * a value x that is not a reference type &T is equivalent to *(x.deref()) or *(x.deref_mut()) as appropriate. Therefore, *counter.lock().unwrap() is equivalent to *(counter.lock().unwrap().deref_mut()).

Finally, we added unwrap() to the two calls to join(); the reason is that join() returns a Result which will be Err if the joined thread panicked. If we do not care if the thread panicked, we can explicitly discard the Result by using std::mem::drop(). (Trivia: how is drop() implemented?3)

1.2.1 Lock poisoning

Why does Mutex::lock return a Result?

Rust's standard library Mutex implements the idea of lock poisoning. If a thread has panicked (terminated with an error) while holding the mutex, then it is likely that it has left the data in an inconsistent state i.e. possibly in a state where some invariants are no longer true. Other threads that try to acquire the mutex should thus be aware of this possibility.

It is valid to just unwrap the result in this case, because if one thread has panicked while holding the mutex, it is likely that other threads will not be able to work with the data anyway. Therefore, unwrapping here will allow panics to propagate across threads.
It is nonetheless still possible to access the data.

2 Scoped threads

While our example now runs, there is still a major difference between the two: our Rust implementation has the counter in the heap!

The problem is that the origin std::thread does not require a join before it is dropped, and therefore it is not possible for the compiler to be satisfied that the threads spawned will not last longer than the main function, outliving counter and making the references captured by the closures dangling references.

What we need is a scoped thread type that will join all threads before main ends, thereby allowing the threads to take closures that do not live for 'static. This functionality was originally provided by a separate library (the Crossbeam crate) but as of Rust 1.63, the standard library implements this as std::thread::scope

Demo 1: Using std::thread::scope

```
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());
}
```

We can simply get rid of references to Arc. Now we just need to have a Mutex living on the stack of our main function; the threads will capture a reference to the mutex.

std::thread::scope will automatically join all remaining threads before it returns, thereby ensuring that main will not end before any threads in the scope end. Thus, again, it is able to capture references to values owned by main.

std::thread::scope returns a Result which must be handled, as usual. The success value is the return value of the outer closure (in this case the unit ()). If any of the threads have panicked, then Err is returned instead with the errors from panicked threads.

Interior mutability

Let us now take a closer look at Mutex<T>. Notice that in the earlier examples, we use a non-mut binding for Mutex and Arc<Mutex<...>>:

```
let counter = Mutex::new(0);
```

Despite the binding being non-mut, we are able to get a mutable reference to the data within the mutex using lock(). Doesn't this seem odd? Is this a flaw in Rust's model?

This is a pattern called **interior mutability**. Having only shared-xor-mutable values is extremely limiting; it would not be possible to mutate any values that are shared by more than one thread. While it is possible to write performant concurrent programs with that restriction using a communication model (like Go), communication is generally less performant than shared memory, and would make Rust unsuitable for being a systems programming language.

Types with interior mutability internally use unsafe code to be able to provide mutable references. Safety is ensured at runtime instead. For Mutex, this is done by ensuring at most one thread can access the mutable reference at any time.

The unsafe guarantee

It is the responsibility of the authors of code using unsafe, and indeed the authors of such types with interior mutability, to ensure that their code only gives out &mut references when it is safe to do so (i.e. in a way that will not lead to data races or memory bugs); the compiler is not able to verify that their code is correct.

Users of libraries (like us) can generally rely on the fact that as long as we do not have unsafe directly in our code, we are safe from data races and memory bugs. This is a guarantee provided by the language: if our entirely-safe code has a data race or memory bug, we can be sure that the bug is not in our code, and instead in a library or in the compiler.

The separation between unsafe and safe code also makes it easier to audit libraries: more attention can be paid to unsafe blocks and the code around those blocks, as those are where memory bugs and data races are likely to stem from.

Other than Mutex, there are other types providing interior mutability in Rust, such as Cell, RefCell, and the atomic types. In our case, we can easily use an atomic type in place of a Mutex<i32>.

Task 2: Atomics in Rust

```
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));
}
```

This is a straightforward replacement of the Mutex<i32> with an AtomicI32.

The memory orderings are the same as those we have seen in C++. In this case, the implicit join before thread::scope returns ensures that we will read the counter only after the threads have incremented them.

Demo 2: Static items

This example is still not the same as the initial C++ example, which had the counter as a global variable. This is possible in Rust too.

```
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));
}
```

We can now do away with std::thread::scope threads, and go back to the original std::thread usage.

Global variables (static items) in Rust are immutable; to modify them, a type with interior mutability is required. This is because, naturally, static items are assumed to be shared, and therefore they cannot be directly mut. Also, a global variable must be of a type that implements the Sync trait, to ensure that the type is safe to be accessed by multiple threads.

(Optional) Rayon

Rayon is a data parallelism library that makes it easy to convert sequential computations into parallel computations. This can speed up your program with relatively little effort!

Let us first take a look at a simple use of Rust's normal Iterator trait to sum a range of numbers.

```
fn magic_sum(from: u128, to: u128) -> u128 {
    (from..to).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()));
}
```

Task 3: Parallelising with Rayon

```
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()));
}
```

L10 - Asynchronous Programming in Rust

Closures
- Can be called like a function
  - But they are not just simple functions
- Can be passed as parameters and returned to and from functions

Overheads of threads
- Context switching cost: threads give up the rest of the CPU time slice when **blocking** functions are called, and a switch happens
  - Registers get restored, virtual address space gets switched, cache gets stepped on, etc.
  - Big cost for high-performance situations (servers) with multiple threads, each handling a connection
- Memory overhead
  - Each thread has its own stack space that needs to get managed by the OS
  - Trying to have 5000 concurrent connections?
    - 5000 threads = 5000 stack segments = 40GB at 8MB/stack!

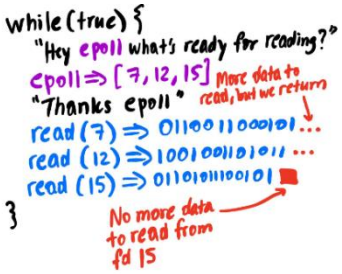
```
fn main() {
    let listener: TcpListener = TcpListener::bind(addr: "127.0.0.1:25565").unwrap();
    for stream_res: Result<TcpStream, Error> in listener.incoming() {
        let mut stream: TcpStream = stream_res.unwrap();
        thread::spawn(move|| {
            let mut str: String = String::new();
            stream.read_to_string(buf: &mut str).unwrap();
        });
    }
}
```

The read() system call can block! (Network connection slow, malicious client, etc).

Mitigate the disadvantages of threads
- Context switches are expensive
  - Use lightweight threads (green threads)
    - Usually they need a runtime (like in Go)
- Is there a way we can have concurrency with less penalties?
  - Non-blocking I/O

Introducing non-blocking I/O
- For example, the read() sys call would block if there is more data to be read but not available
  - Thread gets pulled off the CPU and it cannot do anything else in the meantime
- Instead, we could have read() return a special error value instead of blocking
  - If the client hasn't sent anything yet, the thread can do other useful work, e.g. reading from other descriptors
- **Non-blocking I/O enables concurrency with one thread**!

Non-blocking I/O
- **epoll** is a kernel-provided mechanism that notifies us of what file descriptors are ready for I/O
- Scenario: we are a server having conversations with multiple clients at the same time (multiple fds)
- Epoll notifies the thread when a client said something
- read() from each of those file descriptors, continue those conversations
- Rinse and repeat -> event loop



State management
- Key problem: manage the state associated with each conversation
- Imagine trying to cook 10 dishes at the same time. Need to remember...
  - How long each thing has been on the stove
  - How long things have been in the oven
  - How long things have been marinating for
  - What the next step is for each dish

Actual applications:
• Was I waiting for the client to send me something, or was I in the middle of sending something to the client?
• What was the client asking for before I got distracted?
• Charlie the Client asked me for her emails, but I needed to get them from Dan the Database
• Now Dan the Database responded with some info, but I can't remember what I was supposed to do with it
• Managing one connection in each thread is easy because each conversation is an independent train of thought

Non-blocking I/O is nice in theory, but managing state seems hard!!!

- Rust (and a handful of other languages) take state management to the next level
- **Futures** allow us to keep track of in-progress operations along with associated state, in one package
  - Think of a future as a helper friend that oversees each operation, remembering any associated state

Futures
- Represents a value that will exist sometime in the future
- Calculation that hasn't happened yet
  - Is probably going to happen at some point
  - Just keep asking
- Event loop = runtime for Futures
  - Keeps polling Future until it is ready
  - Runs your code whenever it can be run
  - User-space scheduler for future

Rust: zero-cost abstraction
- Code that you can't write better by hand
  - Abstraction layers disappear at compile-time
- Example: iterators API is a zero-cost abstraction
  - in release mode, the machine code
    - One can't tell if it was done using iterators or implemented by hand
    - Compiler optimize the cache locality that would be difficult to implement by hand

Futures.rs – zero-cost abstraction for futures
- The code in the binary has no allocation and no runtime overhead in comparison to writing it by hand (non-assembly code)
- Building async state machines
- The Future trait
- Event reactor

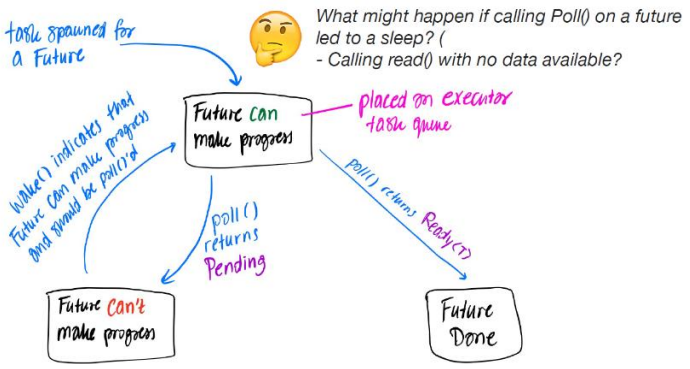
The Future trait
- The executor thread should call poll() on the future to start it off
- It will run code until it can no longer progress

```
trait Future {
// This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
}
enum Poll<T> {
    Ready(T),
    Pending,
}
```

The Future trait
- The executor thread should call poll() on the future to start it off
- It will run code until it can no longer progress
  - If the future is complete, returns Poll::Ready(T)
  - If future needs to wait for some event, returns Poll::Pending, and allows the single thread to work on another future

- When poll() is called, Context structure passed in
- Context includes a wake() function
  - Called when future can make progress again
  - Implemented internally using system calls)
- After wake() is called, executor can use Context to see which Future can be polled to make new progress

Executors
- An executor loops over futures that can currently make progress
  - Calls poll() on them to give them attention until they need to wait again
  - When no futures can make progress, the executor goes to sleep until one or more futures calls wake()
- A popular executor in the Rust ecosystem is Tokio
  - Wraps around mio.rs and futures.rs
- Executors can be single threaded or multi-threaded
- Running on a multiple cores machine
  - Futures can truly run in parallel
  - Need to protect shared data using synchronization primitives (although the ownership model kind of already forces you to do this anyways)



Futures should not block!
- If code within a future causes the thread to sleep, the executor running that code is going to sleep
- Asynchronous code needs to use non-blocking versions of everything, including mutexes, system calls that would normally block, or anything.
- Executor runtimes like Tokio provide these non-blocking implementations for your favorite synchronization primitives

Composition with futures

- Pretty much no one implements futures manually (unless you're a low-level library implementor)
- Instead, futures are combined with various combinators
  - Sequential
  - Concurrently

```
let future = placeOnStove(meat)
    .then(|meat| cookOneSide(meat))
    .then(|meat| flip(meat))
    .then(|meat| cookOneSide(meat));
```

Not great ergonomics

- This code works
  - It's better than manually dealing with callbacks and state machines as you would in C/C++ with interfaces like epoll
- But can we do better?
  - The syntax is a little clunky... too much typing for Rust
  - Code becomes messier as complexity increases
  - Sharing mutable data (e.g. in local variables) can be painful: if there can only be one mutable reference at a time, only one closure can touch that data

Example of poor ergonomics

- Lines 21, 24, 25, 29: asynchronous functions returning Futures
- Lines 27: synchronous (normal) function
- Line 21: complicated syntax

```
21 fn addToInbox(email_id: u64, recipient_id: u64)
22     -> impl Future<Output=Result<(), Error>>
23 {
24     loadMessage(email_id)
25         .and_then(|message|
26             get_recipient(message, recipient_id))
27         .map(|(message, recipient)|
28             recipient.verifyHasSpace(&message))
29         .and_then(|(message, recipient)|
30             recipient.addToInbox(message))
31 }
```

Lines 26: why does get_recipient need to take a message?

- To make this chain of futures work, since the next futures need both the message and recipient as input
- Like a pipeline

Syntactic sugar

Line 41: async function returns Future
Line 44, 45, 47: .await waits for a future and gets its value
Line 45: normal function usage

```
21 fn addToInbox(email_id: u64, recipient_id: u64)
22     -> impl Future<Output=Result<(), Error>>
23 {
24     loadMessage(email_id)
25         .and_then(|message|
26             get_recipient(message, recipient_id))
27         .map(|(message, recipient)|
28             recipient.verifyHasSpace(&message))
29         .and_then(|(message, recipient)|
30             recipient.addToInbox(message))
31 }
```

```
41 async fn addToInbox(email_id: u64, recipient_id: u64)
42     -> Result<(), Error>
43 {
44     let message = loadMessage(email_id).await?;
45     let recipient = get_recipient(recipient_id).await?;
46     recipient.verifyHasSpace(&message)?;
47     recipient.addToInbox(message).await
48 }
```

Async/.await

- An async function is a function that returns a Future
  - Any Futures used in the function are chained together by the compiler
- .await waits for a future and gets its value
  - .await can only be called in an async fn or block
- Structure of the code is similar to what we are used to!
- The Rust compiler transforms this code into a Future with a poll() method
  - Just as efficient as what you could implement by hand

Example: a simple server

- Lines 78, 81, 82: convert any blocking functions to asynchronous versions
- They return Futures

main() now returns a Future

- Futures don't actually do anything unless an executor executes them
- Need to run main() and submit the returned Future to the executor
- Line 73: #[tokio::main] macro submits the future to the executor

```
51 use std::io::{Read, Write};
52 use std::net::TcpListener;
53 use std::thread;
54 fn main() {
55     let listener = TcpListener::bind("127.0.0.1:8080")
56         .unwrap();
57     loop {
58         let (mut socket, _) = listener.accept().unwrap();
59         thread::spawn(move || {
60             let mut buf = [0; 1024];
61             let n = socket.read(&mut buf).unwrap();
62             socket.write_all(&buf[0..n]).unwrap();
63         });
64     }
65 }
```

```
71 use tokio::io::{AsyncReadExt, AsyncWriteExt};
72 use tokio::net::TcpListener;
73 #[tokio::main]
74 async fn main() {
75     let listener = TcpListener::bind("127.0.0.1:8080").await
76         .unwrap();
77     loop {
78         let (mut socket, _) = listener.accept().await.unwrap();
79         tokio::spawn(async move {
80             let mut buf = [0; 1024];
81             let n = socket.read(&mut buf).await.unwrap();
82             socket.write_all(&buf[0..n]).await.unwrap();
83         });
84     }
85 }
```

Async functions generate/return futures

- If you run this function, it will not actually do any work with any messages
- This is still a function and you can still run it...
- But its purpose is now to produce a future that does the stuff that was written inside the function

```
91 async fn addToInbox(email_id: u64, recipient_id: u64)
92     -> Result<(), Error>
93 {
94     let message = loadMessage(email_id).await?;
95     let recipient = get_recipient(recipient_id).await?;
96     recipient.verifyHasSpace(&message)?;
97     recipient.addToInbox(message).await
98 }
```

State management

- There are 5 places where we might be paused, not actively executing:
  - Before anything has happened yet (i.e. Future has been created but not yet poll()ed)
  - Await-ing for loadMessage
  - Await-ing for get_recipient
  - Await-ing for addToInbox
  - Future has completed
- Use an enum to store the state for these possibilities

```
1  enum AddToInboxState {
2  NotYetStarted { email_id: u64, recipient_id: u64 },
3  WaitingLoadMessage {
4      recipient_id: u64, state: LoadMessageFuture },
5  WaitingGetRecipient {
6      message: Message, state: GetRecipientFuture },
7  WaitingAddToInbox {
8      state: AddToInboxFuture },
9  Completed { result: Result<(), Error> },
10 }
```

Attempting to implement poll() for this Future

- Look at the current state and execute the appropriate code from the async fn

```
fn poll() {
    match self.state {
        NotYetStarted(email_id, recipient_id) => {
            let next_future = load_message(email_id);
            switch to WaitingLoadMessage state
        },
        WaitingLoadMessage(email_id, recipient_id, state) => {
            match state.poll() {
                Ready(message) => {
                    let next_future = get_recipient(recipient_id);
                    switch to WaitingGetRecipient state
                },
                Pending => return Pending,
            }
        },
        WaitingGetRecipient(message, recipient_id, state) => {
            match state.poll() {
                Ready(recipient) => {
                    recipient.verifyHasSpace(&message)?;
                    let next_future = recipient.addToInbox(message);
                    switch to WaitingAddToInbox state
                },
                Pending => return Pending,
            }
        },
        ...
```

CS3211

Implications

- Async functions have no stack
  - Sometimes called **stackless** coroutines
  - The executor thread still has a stack (used to run normal/synchronous functions), but it isn't used to store state when switching between async tasks
  - All state is self contained in the generated Future
- No recursion
  - The Future returned by an async function needs to have a fixed size known at compile time
- Rust async functions are nearly optimal in terms of memory usage and allocations
  - Low overhead: the performance is as good as (or possibly better) what you could get tuning everything by hand

Usage of async code

- Taking a step back - problems to solve:
  - Memory usage from having so many stacks
  - Unnecessary context switching cost
- Async code makes sense when...
  - You need an extremely high degree of concurrency
    - Not as much reason to use async if you don't have that many threads
  - Work is primarily I/O bound
    - Context switching is expensive only if you're using a tiny fraction of the time slice
    - If you're doing a lot of work on the CPU for an extended period of time, you might prevent the executor from running other tasks

Similar tools in other languages

- Rust lets us write asynchronous code in the synchronous style that we're used to
- Javascript: very similar toolbox with Promises and async/await
  - Involves much more dynamic memory allocation, not as efficient
- Golang: goroutines are the asynchronous tasks, but unlike Rust they are not stackless
  - Resizable stacks - possible because Go is garbage collected
  - Runtime knows where all pointers are and can reallocate memory
- C++20 just got stackless coroutines
  - Still lots of sharp edges, may want to wait for more libraries to make this easier to use

Summary

- Never block in async code!
  - Asynchronous tasks are cooperative (not preemptive)
- You can only use await in async functions
- Rust won't let you write async functions in traits
  - You can use a crate called async-trait though

Tutorial 9: Asynchronous Programming in Rust

In this tutorial, we look at two versions of a simple TCP line echo server:

- A straightforward threaded implementation that spawns a new thread per client,
- An implementation using Rust Futures and async/await, using tokio.

This program is simple: it should listen for and accept TCP connections, then read and echo back every line (every string ending in \n, including the \n) that it receives. Only complete lines should be sent back. If the server receives e.g. "abc\ndef", only "abc\n" should be sent back. "def" should be buffered until the next \n is received.

The way a network server like this is implemented significantly impacts the maximum number of concurrent clients the server can support given a fixed amount of memory and compute power. (The limiting factor in our case is memory.) It is left as an exercise for the reader to determine the maximum concurrent clients as well as the performance (in terms of lines echoed per second or bytes returned per second) of each variant.

Demo 1: Threaded Version

```
use std::io::prelude::*;
use std::io::BufReader;
use std::net::{SocketAddr, TcpListener, TcpStream};
use std::thread;

fn handle_client(stream: TcpStream) -> std::io::Result<()> {
    let mut reader = BufReader::new(stream);
    let mut buf: Vec<u8> = Vec::new();
    loop {
        let size = reader.read_until(b'\n', &mut buf)?;
        if size == 0 || buf[size - 1] != b'\n' {
            break;
        }
        reader.get_mut().write_all(&buf[..size])?;
        buf.clear();
    }
    Ok(())
}

fn main() -> std::io::Result<()> {
    let port = std::env::args()
        .nth(1)
        .map(|s| s.parse().unwrap())
        .unwrap_or(50000u16);
    let listener = TcpListener::bind(SocketAddr::from(([127, 0, 0, 1],
                                     port)))?;
    loop {
        let (socket, _) = listener.accept()?;
        thread::spawn(move || {
            eprintln!("Accepted connection");
            std::mem::drop(handle_client(socket));
            eprintln!("Connection ended");
        });
    }
}
```

Std::mem:drop is used to ignore the Result returned from handle_client. We do not unwrap as we do not want the server to terminate if one client thread experiences an error.

In handle_client, we wrap the TcpStream in a BufReader which provides us the read_until convenience method, which keeps reading data from the stream until a b'\n' (equivalent to a u8 0xA) is received, and then pushes all the received data into the buf we provide.

Next, we check if read size is zero or the last byte read is not a \n; this indicates that the client has disconnected. If so, we just exit.

If not, this is a complete line, and we just write the entire line to the socket. Because we passed ownership of the socket to the BufReader, we use the BufReader's get_mut method to get a mutable reference to the socket, and then write through that reference. Finally, we clear the buffer, and loop.

That's it! This is a straightforward implementation of an echo server. Unfortunately, it spawns one thread per client, so it isn't really that efficient in terms of memory, since a full thread and stack will be allocated every time a client connects. (On most x86_64 Linux systems, the default thread stack size is 8MB, although this can be adjusted.)

tokio version: async/await

Instead of spawning a new thread for each client and blocking on network I/O, let's instead try a different approach focued on **asynchronous programming** and **non-blocking I/O**. We can break up server initialisation and client connections into different states (bind, accept, read, write), unpausing and transitioning between states for each client only when there is a corresponding I/O event. Implementing this manually however is tedious, as we would have to essentially end up writing a state machine, and we would need to manually store any data that needs to be kept between each state of the state machine.

This is, fundamentally, what async/await helps us do: it lets us write our non-blocking programs in a way that looks sequential, while the compiler does the menial heavy-lifting of transforming it into an efficient state machine for us.

Rust's implementation of futures is quite unique compared to other languages. For one, all that is shipped with the language and standard library is a single Future trait, the async and .await syntax, and compiler support to transform async and .await into efficient state machines. It does not come with methods to do I/O nor an asynchronous execution engine that actually makes the futures run. Instead, following Rust's philosophy of keeping the standard library relatively lean, that is left to external libraries.

The other big difference is that Rust futures are poll-based. In other words, a Future does not represent a computation, but rather a Future is the computation. By polling the future, the computation is driven to completion; without polling the future, nothing happens. This is unlike e.g. JavaScript promises, where simply creating a promise will cause it to be run to completion by the JavaScript engine, even if it is never awaited by anything else.

In Rust, there are currently two major async runtimes, namely Tokio and async-std. Tokio is slightly more popular and generally has had more time to develop. At the moment, unfortunately, many async libraries still require you to use the same runtime they depend on. Tokio generally has the larger ecosystem.

Task 1: Making it async

Let us use Tokio for our implementation. Initialise a new Cargo project and add the dependency:

[dependencies]
tokio = { version = "1", features = ["full"] }

features = ["full"] is part of Cargo's 'features' feature, which allows libraries to make certain features optional, thereby reducing compile time and possibly final binary size. For our case, we will just enable all Tokio features.

Now, start from the code snippet from demo 1, and make it async. Hints:
Start off with #[tokio::main].
To have a future run to completion without awaiting it, use tokio::spawn.
Async/await in general: async/await

```
use std::net::SocketAddr;

use tokio::io::{AsyncBufReadExt, AsyncWriteExt, BufReader};
use tokio::net::{TcpListener, TcpStream};

async fn handle_client(stream: TcpStream) -> std::io::Result<()> {
    let mut reader = BufReader::new(stream);
    let mut buf: Vec<u8> = Vec::new();
    loop {
        let size = reader.read_until(b'\n', &mut buf).await?;
        if size == 0 || buf[size - 1] != b'\n' {
            break;
        }
        reader.get_mut().write_all(&buf[..size]).await?;
        buf.clear();
    }
    Ok(())
}

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let port = std::env::args()
        .nth(1)
        .map(|s| s.parse().unwrap())
        .unwrap_or(50000u16);
    let listener = TcpListener::bind(SocketAddr::from(([127, 0, 0, 1],
port))).await?;
    loop {
        let (socket, _) = listener.accept().await?;
        tokio::spawn(async move {
            eprintln!("Accepted connection");
            std::mem::drop(handle_client(socket).await);
            eprintln!("Connection ended");
        });
    }
}
```

The only point of explanation here is the async block in the loop; this async block evaluates to a Future that we can return or pass to another function (in this case, our executor). It is essentially akin to having an async fn or async closure1 that is immediately invoked. Just like a closure, it can capture values from its environment, and it can also take the move keyword. In this case, we want the Future to take ownership of socket, so we use the move keyword.

Configuring the Tokio runtime

While the Tokio defaults (multi-threaded runtime using as many threads as there are cores) are usually sufficient, you may sometimes want to tweak some settings.

For example, you may want to want to manually set the number of worker threads. You can do this using the tokio::main macro:

#[tokio::main(worker_threads = 2)]
async fn main() { ... }

Alternatively, if you need to programmatically set the number of threads, you can use the Builder, without using tokio::main:

```
    let threaded_rt = tokio::runtime::Builder::new_multi_thread()
        .worker_threads(2)
        .enable_all()
        .build()?;
…
threaded_rt.block_on(async {
        let listener = TcpListener::bind(SocketAddr::from(([127, 0, 0,
1], port))).await?;
        loop {
            let (socket, _) = listener.accept().await?;
            tokio::spawn(async move {
                eprintln!("Accepted connection");
                std::mem::drop(handle_client(socket).await);
                eprintln!("Connection ended");
            });
        }
    })
```

There are a few minor differences. First, our main function is now a normal function, not an async function. We manually create a runtime using tokio::runtime::Builder, set the number of worker threads, and then call enable_all, which is needed for Tokio to initialise its internal bookkeeping, etc., to run I/O and time operations.

Secondly, since our main function is not itself async anymore, the I/O part is put into a separate async block, and we explicitly pass the resulting Future to the runtime we created earlier.

This is exactly what the tokio::main macro does for us.

Demo 2: Single-threaded runtime

Sometimes, you may want to run asynchronous I/O without using more than one thread; in such situations you can use the current_thread runtime:
#[tokio::main(flavor = "current_thread")]

Alternatively, you can instantiate it manually as well:

```
let rt = tokio::runtime::Builder::new_current_thread()
    .enable_all()
    .build()?;
```

Try to run this with multiple clients while observing the process in htop. You will see that no additional threads are created aside from the single main thread, but we are still able to handle multiple clients. (And we didn't need to write a state machine by hand!)

Task 2: Synchronising futures with Tokio (and channels!)

Just like how we have synchronisation primitives to synchronise threads, Tokio also provides synchronisation primitives that work with futures. These primitives, however, do not cause the running thread to block; rather, as you would expect, they work with .await and cause the awaiting task to be paused until the lock is released, the barrier is reached, etc.

Recall the C++ barrier-and-semaphore solution to the H2O problem from tutorial 6:

```
use std::sync::Arc;

use futures::future::join_all;
use tokio::sync::{mpsc, Barrier, Mutex, 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.unwrap();
    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 (s, r) = mpsc::channel(2);
    let r = Arc::new(Mutex::new(r));

    let hydrogens =
        (0..200).map(|i| tokio::spawn(hydrogen(i, barrier.clone(),
h_sem.clone(), s.clone())));
    let oxygens = (0..100).map(|i| tokio::spawn(oxygen(i,
barrier.clone(), r.clone())));

    let join_handles = Iterator::chain(hydrogens,
oxygens).collect::<Vec<_>>();

    std::mem::drop(s);
    std::mem::drop(r);

    join_all(join_handles).await;
}
```

For the base problem, we simply need to instantiate two Semaphores and a Barrier. The Semaphores and Barrier must be shared among the tasks using an Arc, as usual. (Rc is not sufficient because Futures could move threads during their lifetime, therefore our Futures must be Send.)

You will likely face the problem of the program ending before all molecules have been formed. To solve that, we must collect the JoinHandles of all the tokio::spawn invocations, join then using future::join_all, and then await the joined future in the main task.

To solve the challenge problem, we make O atoms leaders. Then, we instantiate an MPSC channel; hydrogen atoms will send their IDs to O atoms, which will print. The MPSC Sender can simply be clone()d for each H task, but the Receiver, this being a single-consumer channel, cannot be cloned, but fittingly for this problem, we can just replace the O semaphore with a Mutex and put the Receiver behind it. As usual, the Mutex must be shared among the O tasks with an Arc.