CS3211: Parallel and Concurrent Programming

by: Jiahao

concurrency

two or more separate activities happening at the same time

- specifically, a single system performing multiple independent activities in parallel, rather than sequentially
  - earliest instance of multitasking in OS
- enabled using multiple processes and/or threads

⋆ determined by how the code written

illusion of concurrency

achieving concurrency through **task switching**

- starting, executing, and finishing in an overlapping fashion
- incurs **context switch** cost

true concurrency

with multiple cores/processors, machines can actually run tasks in parallel

- each core can still perform interleaving

hardware threads

dictates the amount of true concurrency

- processor can contain many physical cores
- core can support multiple hardware threads (SMT)
- physical cores may be mapped to logical cores that represent the number of processes that can be executed

parallelism

two or more tasks can run (execute) simultaneously, at the exact same time

⋆ determined by the number of cores of a machine

- tasks do not only make progress, but they also execute simultaneously
- parallelism $\subseteq$ concurrency

processes

- each process owns its own memory space
- both exceptions and interrupts introduce overhead (due to context switching)

exceptions

occurs when executing a machine level instruction

- e.g. overflow, underflow, division by zero, illegal memory access, misaligned memory access
- have to execute an exception handler

synchronous: occur due to program execution

interrupts

occurs when an external events interrupt the execution of the program

- usually hardware related: timer, mouse movement, keyboard pressed, etc.
- have to execute an interrupt handler

asynchronous: occur independently of program execution

disadvantages

creating a new process is costly: overhead of system calls and allocating memory

communicating between processes is costly: communication is enabled through the OS via syscalls

threads

defines a sequential execution stream within a process

- own stack and registers
- rest of memory space shared
- each thread shares the memory space with other threads of the same process
- extension of process model
- process contains multiple independent threads
- no OS to communicate between threads ⇒ less overhead
- thread generation is faster than process generation
  - no copy of address space

⋆ different threads of a process can be assigned to run on different cores of a multicore processor

types of threads: usually mapped 1 : 1

1. **user-level thread:** created by libraries
2. **kernel thread:** managed by OS

POSIX threads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *print_message_function(void *ptr);

int main() {
  pthread_t thread1, thread2;
  char *message1 = "Thread 1";
  char *message2 = "Thread 2";
  int iret1, iret2;

  iret1 =
      pthread_create(
        &thread1,
        NULL,
        print_message_function,
        (void *)message1
      );
  iret2 =
      pthread_create(
        &thread2,
        NULL,
        print_message_function,
        (void *)message2
      );

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  printf("Thread 1 returns: %d\n", iret1);
  printf("Thread 2 returns: %d\n", iret2);
  exit(0);
}

void *print_message_function(void *ptr) {
  char *message;
  message = (char *)ptr;
  printf("%s \n", message);
}
```

mutual exclusion

- use mutual exclusion to synchronize access to shared resources
- lock the mutex to access the shared resource, and unlock once done using
- structure code to use mutexes to avoid conflicting access by multiple threads to shared resources
- mutex is locked by one thread
  - only that thread can hold the lock

mechanisms

locks → primitive, minimal semantics, used to build others

semaphores → basic, easy to get the hang of, but hard to program with

monitor → 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

problems in concurrent programs

race conditions

occurs when the timing or order of events affects the outcome of executing concurrent code

⋆ not emphasized in CS3211

- not all race conditions are bad (i.e. don't affect the correctness of the code)
- avoid race conditions that affect the correctness of code → use critical sections
  - creates mutual exclusion

data race

conditions:

1. two concurrent threads (or processes) access a shared resource (memory address) without any synchronization
2. at least one thread modifies the shared resource

- requires controlled access to shared resources
  - synchronize access to any shared data structure

deadlock

there is an execution where no progress is made when it should be possible

- covers both deadlock and livelock

occurs when threads...

- compete for access to limited resources
- are incorrectly synchronized
- need multiple locks to access some shared resources

conditions

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 $[P_1, P_2, ..., P_n]$ such that P_1 is waiting for P_2 and so on

handling

- ignore it
- prevention
  - ensure locks are acquired in the same order
  - avoid connection of threads in circular fashion
- avoidance
- detection and recovery
  - look for cycle in dependencies

starvation

thread is prevented from making progress because some other threads have the resource it requires, and the initial thread does not get the chance to acquire the resource

side effect of the scheduling algorithm: high-priority process prevents low-priority process from running or one thread always beats another when acquiring a lock

pros vs cons of concurrency

pros	cons
• separation of concerns • performance by taking advantage of hardware and having an optimization strategy	• concurrency issues • maintenance difficulties due to complicated code and hard to debug • threading overhead due to stack and context switching

concurrent and parallel programming challenges

1. finding enough parallelism (Amdahl's Law)
2. granularity of tasks
3. locality
4. coordination and synchronization
5. debugging
6. performance monitoring

program execution with threads

1. decomposition of computations
2. assigning tasks to threads
3. orchestration
   - structuring communication
   - adding synchronization to preserve dependencies
   - organizing data structures in memory
   - scheduling tasks
4. mapping of threads to physical cores

task parallelism

divide work into tasks to make the threads specialists

- same type of tasks assigned to the same thread (aka pipeline)
- divide the work by task type to separate concerns
- use task pools and the number of threads to serve the task pool

data parallelism

dividing data into chunks and having each thread take a chunk

C++ history

- 1998: original C++ standard published
  - no support for multithreading
  - use external libraries to manage threads in C/C++ programs (`pthread`)
- 2011: C++11 standard published
  - new "train model" of releases
  - introduced multithreading
- 2014
- 2017
- 2020
- 2023

C++98

- does not acknowledge the existence of threads
- effects of language elements assume a sequential abstract machine
  - no memory model
- multithreading dependent on compiler-specific extensions
  - e.g. C APIs and Microsoft Windows APIs or application frameworks like Boost and ACE

C++11

- formalized multithreading
- allowed writing portable multithreaded code with guaranteed behavior
  - multithreading without relying on platform-specific extensions
- **thread-aware memory model**
- include classes for managing threads, protecting shared data, synchronization between threads, low-level atomic operations
- support use of concurrency to improve application performance
  - take advantage of increased computing power
  - **low abstraction penalty:** C++ classes wrap low-level facilities (i.e. hardware calls and atomic operation library)

threading in C++

- every program has at least one thread
  - runs `main()`
  - started by C++ runtime

creating a thread

```
#include <iostream>
#include <thread>

void hello() {
  std::cout << "Hello world\n";
}

int main() {
  std::thread t(hello);
  t.join();
}
```

starting threads

most vexing parse

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

- above does not start the thread, instead, it declares `my_thread` as a function that takes in a function pointer and returns a `std::thread`
- place another set of parentheses around `background_task()` to avoid this

thread with a function

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

thread with a function object

```
void do_some_work();

class background_task {
public:
  void operator()() const {
    do_some_work();
  }
}

background_task f;
std::thread my_thread(f);
std::thread my_thread((background_task()));
std::thread my_thread{background_task()};
```

thread with a lambda expression

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

waiting

wait for a thread to finish

- **`join()`**: join exactly once
- **`joinable()`**: check that a thread can be joined

⋆ remember to join even if there is an exception

```
void f() {
  std::thread t(foo);
  try {
    // something that can raise an exception
  } catch() {
    // MUST JOIN EVEN IN CATCH
    t.join();
    throw;
  }

  t.join();
}
```

without `join()`: `std::terminate` must be called, killing the program

- observe non-zero return value

detaching

detach a thread from its parent

- **`detach()`**
- ensure that local variables passed to the thread are accounted for
- once detached, the thread is not joinable

use after free: local variables may end their lifetimes before a thread ends

```
std::thread t(foo);
t.detach();
assert(!t.joinable());
```

passing arguments to a thread

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

- the above passes the arguments by value

```
void f(int i, std::string const& s);
void oops(int some_param) {
  char buffer[1024];
  sprintf(buffer, "%i", some_param);
  std::thread t(f, 3, buffer);
  t.detach();
}
```

- the above may have a use-after-free
  - the lifetime of **`buffer`** ends after **`oops()`** returns
  - when **`buffer`** is passed to the thread, **`std::thread`** copies the arguments into an internal storage to be used later
  - the default behavior for **`std::thread`** is to **pass-by-value**, so the **`buffer`** pointer is passed
  - since the thread may start after **`oops()`** returns, there is a chance that the thread only uses the **`buffer`** pointer AFTER the lifetime of **`buffer`**

```
void f(int i, std::string const& s);
void oops(int some_param) {
  char buffer[1024];
  sprintf(buffer, "%i", some_param);
  std::thread t(f, 3, std::string(buffer));
  t.detach();
}
```

- the above fixes the possible use-after-free since it creates the **`std::string`** for **`std::thread`**'s internal storage, which is not affected by the **`buffer`** going out of scope

```
void update_data(id i, data& d);
void oops(id i) {
  data d;
  std::thread t(update_data, i, d);
  // ...
}
```

- the above passes the **`data`** by value, which means it is copied
  - any changes to **`data`** will not be seen within **`std::thread`** (since it holds a copy)

```
std::thread t(update_data, i, std::ref(d));
```

- the above fixes the issue as we explicitly specify that we are passing a reference to **`data`**
  - similar to what we did with **`buffer`**

lifetime

lifetime of an object begins when

- storage with the proper alignment and size for its type is obtained
- its initialization (if any) is complete

lifetime of an object ends when

- non-class type ⇒ object is destroyed
- class type ⇒ destructor call starts
- storage which the object occupies is released or reused by an object not nested within it

lifetime of a reference

- begins when initialization completes
- ends like a scalar object

dangling reference: lifetime of referred object ends before lifetime of reference

ownership in C++

owner: object containing a pointer to an object allocated by **`new`** for which a **`delete`** is required

- every object on the free store (i.e. heap) must have exactly one owner
  - not enforced, but advised
- based on use of constructors and destructors
  - **scoped objects:** destruction is implicit at scope exit
  - **objects on free store:** use **`delete`**
    - alternative: **`malloc()`** and **`free()`**

ownership of a thread

- **`std::thread`** instances own a resource
  - manage a thread of execution
- instances of **`std::thread`** are movable (pass ownership) and not copyable
  - ownership can also be transferred by returning the thread from a function

```
void foo();
void bar();

std::thread t1{foo};
std::thread t2{std::move(t1)};

t1 = std::thread(bar);
std::thread t3;
t3 = std::move(t2);
// this fails because t3 already manages something
t1 = std::move(t3);
```

RAII (Resource Allocation Is Initialization)

binds lifecycle of a resource that must be acquired before use to the lifetime of an object

- resource e.g. allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection
- owner must typically destroy or pass ownership

synchronizing concurrent accesses

- all shared data is read-only ⇒ no problem
- modifying shared data has challenges
- programs must be designed to ensure that changes to a chosen data structure are correctly synchronized among threads
  - data structures are immutable or protect data using some locking

structure of program data

- 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 **`int`** and **`char`** occupy exactly one memory location, regardless of size
- adjacent bit fields are part of the same memory location

data races

two accesses to a single memory location from separate threads

- no enforced ordering between accesses

conditions: one or both accesses are not atomic AND one or both are writes

conflicting expression evaluations:

1. one modifies memory or starts/ends the lifetime of an object in a memory location
2. other reads or modifies the same memory location or starts/ends the lifetime of an object occupying storage that overlaps with the memory location

data race unless...

- both evaluations execute on the same thread or signal handler
- both evaluations are atomic operations

- one evaluation happens-before another

results in undefined behavior: no restrictions on the behavior of the program
- caused by data races, memory accesses outside of array bounds, signed integer overflow, etc.

avoidance:
- locks
- ordered atomics
- transactional memory

invariants

statements that are always true about a particular data structure
- often broken during an update on the data structure
  - but must hold before and after the update
- used to reason about program correctness

race conditions: other threads working on data while invariant is broken

dealing with race conditions

problematic execution sequence: occurring increases when
- there is high load in the system
- the operation is performed more times

simple solution: wrap data (structure) with protection mechanism
- ensure only the thread performing a modification can see the intermediate states while the **invariants are broken**

mutex

*mut*ual *ex*clusion

serialization: threads take turns accessing the data protected by the mutex

construction: `std::mutex mutex_name`

locking: `.lock()`

unlocking: `.unlock()`

manual locking/unlocking: requires `.unlock()` for every code path, including exceptions
- not recommended

RAII for mutexes: `std::lock_guard{mutex}`
- locks supplied mutex on construction and unlocks on destruction

```
#inlcude <mutex>
#include <list>
std::list<int> some_list;
std::mutex lock;

void add_to_list(int new_value) {
  std::lock_guard<std::mutex> guard{lock};
  some_list.push_back(new_value);
}
```

common practice: group mutex and protected data together in a class over global variables

fine grained locking:
- mutex per node
- mutex on dummy nodes
- mutex on master entry

passing references with mutexes

```
class data_wrapper {
public:
  some_data data;
  std::mutex m;

  void process_data(Function func) {
    std::lock_guard<std::mutex> l{m};
    func(data);
  }
}

some_data* unprotected;
void malicious(some_data& data) {
  unprotected = &data;
}

data_wrapper x;
void foo() {
  x.process_data(malicious);
  // Access here happens AFTER lock is released
  unprotected->foo();
}
```

unprotected access: access to `some_data data` is performed AFTER lock is released from `process_data()`

recommendation: don't pass pointers or references to protected data outside the scope of the lock
- returning from a function
- storing in externally visible memory
- passing them as arguments to user-supplied functions

avoiding deadlocks

1. avoid nested locks: or use `std::lock` to lock multiple mutexes at once
2. avoid calling user-supplied code while holding a lock
3. acquire locks in a fixed order

hand-over-hand locking

holding a lock on the current node while acquiring the lock on the next node
- ensures that the `next` pointer is not modified

potential deadlock:
1. $T1$ acquires lock for $N1$
2. $T2$ acquires lock for $N2$ and $N3$
3. $T1$ tries to acquire lock for $N2$ (currently held by $T2$)
4. $T2$ tries to acquire lock for $N1$ (currently held by $T1$)

other types of locks

`std::unique_lock`: does not always own the mutex that it is associated with
- allows for locking the mutex later using `std::defer_lock`
- only one thread can lock the mutex

```
std::unique_lock<std::mutex>
  lock_prev{prev, std::defer_lock};
std::unique_lock<std::mutex>
  lock_next{next, std::defer_lock};
// ...
std::lock(lock_prev, lock_next);
```

- can also transfer ownership by
  1. returning `std::unique_lock` from function
  2. calling `.unlock()`

`std::lock()`: locks one or more mutexes at once without risk of deadlock
- uses deadlock detection algorithm to release partial locks
- use `std::adopt_lock` to indicate to `std::lock_guard` objects that mutexes are already locked
  - adopt ownership of existing lock on mutex rather than attempting to lock mutex in the constructor

```
std::lock(prev, next);
// Adopt the locks from above
std::lock_guard<std::mutex>
  lock_prev{prev, std::adopt_lock};
std::lock_guard<std::mutex>
  lock_next{next, std::adopt_lock};
```

`std::scoped_lock`: accepts and locks a list of mutexes
- locks in the same way as `std::lock()`

```
std::scoped_lock<std::mutex, std::mutex>
  guard{prev, next};
```

semaphores

counting semaphore: `std::counting_semaphore<int> count;`
- `count{n}` where `n` is the starting value
- `release()` increments by 1
- `acquire()` blocks while count is 0; decrements by 1 once acquired
- `try_acquire()` will attempt the acquisition
  - return `false` if blocked

concurrent queue & blocking dequeue

refer to tutorial 1 for implementation details

synchronizing concurrent actions

ensuring that actions run in some order even if executed concurrently

task dependency: thread A waiting for thread B to complete a task

busy waiting: wasteful since thread A is blocked waiting
- thread A keep checking a flag in shared data (protected by a mutex)
- thread B set the flag when completed

wait and sleep: while waiting for flag, sleep if flag is not available
- less wasteful

```
bool flag;
std::mutex m;
void wait() {
  std::unique_lock<std::mutex> l{m};
  while(!flag) {
    l.unlock();
    // std::this_thread
    // std::chrono::milliseconds
    this_thread::sleep_for(milliseconds(100));
    l.lock();
  }
}
```

condition variable

associated with an event or other condition and one or more threads can wait for that condition to be satisfied

⋆ optimization over busy waiting

- when condition is satisfied, thread notifies one or more of waiting threads
- notified threads wake up and continue processing
  - ensure that threads are notified during exception cases too

`std::condition_variable`: works with **`std::mutex`**

- simpler, lightweight, less overhead
- condition not satisfied ⇒ **`wait()`** unlocks mutex and puts the thread in blocked/waiting state
  - until **`notify_one()`** or **`notify_all()`** called
  - reacquires lock before proceeding

```
std::mutex m;
std::queue<int> q;
std::condition_variable cond;

void first() {
  while (true) {
    int a = produce();
    {
      std::lock_guard<std::mutex> l{m};
      q.push(a);
    }
    cond.notify_one();
  }
}

void second() {
  while (true) {
    std::unique_lock<std::mutex> l{m};
    // equivalent to
    // while (q.empty) {
    //   cond.wait(l);
    // }
    cond.wait(l, []{ return !q.empty() });
    int data = q.front();
    q.pop();
    l.unlock();
    process(data);
  }
}
```

during `wait()`: condition variable

- check supplied condition any number of times
  - avoid side effects for condition check
- checks condition with mutex locked
- returns immediately iff the condition returns true

spurious wake: waiting thread reacquires the mutex and checks the condition, but not in direct response from **`notify_*()`**

waiting with time limit: blocking calls block indefinitely, suspending the thread until the event occurs

- **`wait_for()`**: duration based timeout
- **`wait_until()`**: absolute timeout (fixed time)

`std::condition_variable_any`: works with anything mutex-like

- potentially more costs in size, performance, or OS resources

C/C++ code execution

compilation and linking: done by the compiler

1. preprocessor: replaces preprocessor directives (**`#include`** and **`#define`**)
2. compiler: parses pure C++ source code and converts to assembly
3. assembler: assembles code into machine code
4. linker: produces final compilation output from object files produced by compiler
   - pulls other libraries (if any)

loading: specific to the OS

execution: coordinated by the OS

- program gets access to CPU, memories, devices, etc.

concurrency expectations

correctly synchronized program should behave AS IF

- memory operations are actually executed in an order that appears equivalent to some sequentially consistent interleaved execution of the memory operations of each thread
- including the each write appears to be atomic and globally visible simultaneously to all processors

operation reordering: processor might reorder operations while preserving the above requirements

- $W \to W$: reorder operations in a write buffer (e.g. due to cache miss)
- $R \to W$ or $R \to R$: reorder *independent* instructions in an instruction stream (out-of-order execution)

memory model (MM)

provide contract to programmers about what are the visible side effects of their memory operations

- allows processors to optimize operations without affecting program correctness
- essential to make all multithreading facilities work
  - mutex must be aware of MM in C++

use MM to avoid: deadlocks and data races

as-if rule

C++ compiler is permitted to perform any changes to the program as long as:

1. accesses to volatile objects occur strictly according to the semantics of the expression in which they occur
   - they may not be reordered w/r/t other volatile accesses on the same thread
2. at program termination, data written to files is exactly as if the program was executed as written
3. prompting text which is sent to I/O will be shown before the program waits for input

undefined behavior: free from as-if rule

C++

```
int main() {
  int x = 5;
  int y = x + 1;

  return y;
}
```

assembly

```
main:
  mov eax, 6
  ret
```

language level memory models

languages guarantee sequential consistency for data-race-free programs

- compilers will insert the necessary synchronization to cope with the hardware memory model

guarantee: only if every atomic uses sequential consistency

C++ memory model

sequenced-before (sb)

within the same thread, evaluation A *may* be sequenced-before evaluation B (i.e. program order)

- A sb B ⇒ evaluation of A will be complete before the evaluation of B

```
while (!y.load()); // A
if (x.load()) ++z; // B
```

- A sb B
- appear to execute in the order they are written
  - but if the visible side effects are not affected, then it might reorder

synchronizes with (sw)

suitably-tagged atomic store operation (release), W in thread A on a variable x synchronizes-with a suitably-tagged atomic load operation (acquire), R in thread B on x if R reads the value stored by W

- appears between **atomic load and store** operations
- R MUST read from W to be considered a sw relationship
- if W has already been read, then there is no sw relationship with future values

⋆ only on operations on atomic types

A

```
x.store(true)
```

B

```
while (!x.load());
```

- A sw $B \iff x = \text{true}$

A ***B***

A	*B*
`x.store(true)`	`if (!x.load());`

- A might not sw B since B does not wait (using `while`) for $x = \text{true}$

simply happens-before (hb)

A simply happens-before B (regardless of thread) if any:

1. A sb B
2. A sw B
3. A simply happens-before X and X simply happens-before B $\Rightarrow$ transitivity

A	*B*

`x.store(true) // 1`	`// 2`
	`while (!x.load());`
	`// 3`
	`if (y.load()) ++z;`

- 1 hb 2: sw relationship
- 2 hb 3: sb relationship
- 1 hb 3: transitive sw $\rightarrow$ sb relationship

strongly happens-before

specifies which operations see the effects of which other operations

- A strongly happens-before $B \Rightarrow A$ appears to be evaluated before B in all contexts

A strongly happens-before B if any:

1. A sb B
2. A sw B and A and B are sequentially consistent atomic ops
3. A sb $X \wedge X$ hb $Y \wedge Y$ sb B
4. A strongly happens-before X and X strong happens-before B

visible side effects

side-effect A on scalar M (a write) is visible w/r/t value computation B on M (a read) if both are true:

1. A hb B
2. no other side effect X to M where A hb $X \wedge X$ hb B

visible sequence of side-effects: side-effect A is visible w/r/t value computation $B \Rightarrow$ longest contiguous subset of side-effects to M, in modification order, where B does not hb it is known

modification order (MO)

composed of all writes to an object from all threads in the program

- can vary between runs

non-atomic types: programmer's responsibility to ensure MO using synchronization

- data race and undefined behavior are possible

atomic types: compiler's responsibility to ensure necessary synchronization such that all threads agree on the MO for each atomic variable

- does not mean "race-free"

requirements for atomic types:

1. once a thread sees an entry in the MO
   1. subsequent reads from that thread must return that value or later values
   2. subsequent writes from that thread to that object must occur later in the modification order
2. all threads must agree on the MO of each individual atomic
   - may not agree on relative order of op on separate objects

definitions: all modifications to any particular atomic variable occur in a total order that is specific to that atomic variable

1. **write-write coherence:** A modifies atomic M hb B modifies $M \Rightarrow A$ appears earlier than B in the modification order of M
2. **read-read coherence:** A reads atomic M hb B reads M and A comes from write X on $M \Rightarrow B$ is either the value stored by X or value stored by side effect Y on M that appears later than X in the modification order of M
3. **read-write coherence:** A reads atomic M hb B modifies M $\Rightarrow A$ comes from a side-effect X that appears earlier than B in the modification order of M
4. **write-read coherence:** a write X on atomic M hb a read B of $M \Rightarrow B$ takes its value from X or a side-effect Y that follows X in the modification order of M

coherence order

the above refers to the coherence order, which records when each read operation is performed, relative to the writes in the sequence

- each read operation is placed immediately after the write that it reads from

no guarantees about relative order of operations on different atomic variables

atomics

indivisible op always observed fully done from any thread in the system

- low-level synchronization op reduces to 1-2 CPU instructions
- MAY be faster than locking

data loading: initial value or value stored by a modification

non-atomic operations: can be seen half-done by another thread

- composed of atomic operations

`#include <atomic>`: all ops on type is atomic

- compiler translates into atomic machine instructions
- `is_lock_free() == true`: if op is done directly with atomic instructions for all supported hardware

emulating atomicity: using mutex

- performance gains may not materialize

read-modify-writes (rmw): complex operations that involve reading the current value of the atomic variable, performing some operation, and *possibly* writing it back

- `fetch_all`, `fetch_sub`, `exchange`, `compare_exchange_weak`, `compare_exchange_string`
- indivisible regardless of memory order
  - always read the last value in the modification order written before the write associated with the rmw operation

memory order

every atomic operation has an optional memory-ordering argument (from `std::memory_order`)

- specifies how memory accesses, *including non-atomic*, are to be ordered around an atomic operation

`std::memory_order`: default (and strongest) is `seq_cst`

Operation (`memory_order_`)	Store	Load	R-M-W
`seq_cst`	✅	✅	✅
`release`	✅		✅
`acquire`		✅	✅
`consume`		✅	✅
`acq_rel`			✅

reasoning: reason with memory orders as axioms

- every behavior that is not explicitly forbidden by the C++ standard is legal

refer to tutorial 2 slides for memory barrier behavior to contexualize how memory orders work

sequentially consistent ordering

behavior of the program is consistent with a simple sequential view of the world

- all threads must agree on the same order of ops
- ops from the same thread cannot be reordered
- seq-cst store s-w seq-cst load of same variable
  - does not apply to relaxed memory orderings

performance penalty: on weakly-ordered machine with many cores

- requires extensive (and expensive) sync ops among cores

```
std::atomic<bool> x, y;
std::atomic<int> z;
void write_x() { x.store(true); }
void write_y() { y.store(true); }
void read_x_then_y() {
  while(!x.load());
  if (y.load()) ++z;
}
void read_y_then_x() {
  while(!y.load());
  if (x.load()) ++z;
}
int main() {
  x = false;
  y = false;
  z = 0;
```

```
  std::thread a{write_x};
  std::thread b{write_y};
  std::thread c{read_x_then_y};
  std::thread d{read_y_then_x};
  a.join();
  b.join();
  c.join();
  d.join();
  assert(z.load() != 0);
}
```

- either `write_x()` or `write_y` has to happen first ⇒ `assert` never fires

non-sequentially consistently orderings: no single global order of events

- different threads can see different views of the same operation
- only requirement: all threads agree on the modification order of each individual variable

relaxed ordering

⋆ do not participate in synchronizes-with relationships

- guarantee atomicity and modification order consistency
- ops on same variable within single thread obey happens-before relationships
  - ‣ but **no guaranteed order relative to other threads**
- accesses to a single atomic variable from the same thread cannot be reordered
  - ‣ once a thread sees a particular value of an atomic variable (visible side effect), subsequent reads cannot retrieve an earlier value

```
std::atomic<bool> x, y;
std::atomic<int> z;
void write_x_then_y() {
  // 1
  x.store(true, std::memory_order_relaxed);
  // 2
  y.store(true, std::memory_order_relaxed);
}
void read_y_then_x() {
  // 3
  while(!y.load(std::memory_order_relaxed));
  // 4
  if (x.load(std::memory_order_relaxed)) ++z;
}
int main() {
  x = false;
  y = false;
  z = 0;
  std::thread a{write_x_then_y};
  std::thread b{read_y_then_x};
  a.join();
  b.join();
  assert(z.load() != 0);
}
```

- `assert` could fire
- 1 sb 2 ∧ 3 sb 14 ⇒ 1 must run before 2, and 3 must run before 4 (in program order)
- however, the actual modification order of x and y might not follow the program order (since they are relaxed)
  - ‣ ⇒ loads can see the stores out of order if there's no synchronizes with and it's running in different threads

layman interpretation

- there is no happens-before relationship between (1) and (4) because they exist on different threads and their relaxed order means that there are no guarantees in the order between threads.
- even though (1) is sequenced before (2) in the program order, the relaxed memory ordering means that these operations can also be reordered as observed by other threads
- thread B can observe `y` as true but observe `x` as false

levels or reordering

1. **physical reordering:** the CPU may literally execute the instructions in a different order than written in the source code. Modern CPUs use out-of-order execution to optimize performance, so the store to y might physically happen before the store to x despite the source code ordering.
2. **memory subsystem reordering:** even if the CPU executes instructions in the written order, the memory operations may complete in a different order due to cache behavior, write buffers, and the memory hierarchy.
3. **compiler reordering:** the compiler itself can reorder operations during optimization when it's allowed by the memory model.
4. **visibility to other threads:** most importantly, other threads on other CPU cores may observe these stores in a different order than they were written. This is because each CPU core might have its own cache with different timing for when values become visible.

```
std::atomic<int> x;
std::atomic<bool> y;
void foo() {
  x.store(1, std::memory_order_relaxed);
  y.store(true, std::memory_order_relaxed);
  while (!y.load(std::memory_order_relaxed))
    ;
  std::cout << x.load(std::memory_order_relaxed) <<
std::endl;
}
int main() {
  x = 0;
  y = false;
  std::thread a{foo};
  a.join();
}
```

- the above will always return `1` because the program order is preserved within the same thread, even if it uses a relaxed ordering

release-acquire ordering

if an atomic store in thread A is `release` and an atomic load in thread B for the same variable is `acquire`, and the load in B reads a value written by the store in A, then the store in A synchronizes-with the load in B

- all memory writes (both non-atomic and relaxed) that happened-before the atomic store from the point of view of A become visible side-effects in B
  - ‣ once the atomic load completes, B is guaranteed to see everything A wrote to memroy
  - ‣ only held if B returns the value A stored or a value later on

`release` effect: all writes that happened before it is now visible to all other threads

- no read or writes in the current thread can be reordered after the `release`
- creates a "safe group" of actions before the `release` that are guaranteed to happen

`acquire` effect: no read or writes in the current thread can be reordered before the `acquire`

- ensures that no future actions can happen before the `acquire`

release sequence headed by operation A: longest continuous subsequence of the modification order of atomic object M after A on M that consists of:

1. writes performed by the same thread that performed A
2. atomic read-modify-write ops made to M by any thread

intuition:

```
mutex.lock();
// acquire updates from other threads
// updates NOT available to other threads until unlock()
mutex.unlock();
// release updates this thread has made to other threads
// memory updates not available to other threads until
they call lock
```

synchronization:

```
std::atomic<bool> x, y;
std::atomic<int> z;
void write_x_then_y() {
  // 1
  x.store(true, std::memory_order_relaxed);
  // 2
  y.store(true, std::memory_order_release);
}
void read_y_then_x() {
  // 3
  while(!y.load(std::memory_order_acquire));
  // 4
  if (x.load(std::memory_order_relaxed)) ++z;
}
int main() {
  x = false;
  y = false;
  z = 0;
  std::thread a{write_x_then_y};
  std::thread b{read_y_then_x};
  a.join();
  b.join();
  assert(z.load() != 0);
}
```

- `assert` never fires
- 1 sb 2 ∧ 3 sb 4
- 2 sw 3

no synchronization:

```
std::atomic<bool> x, y;
std::atomic<int> z;
void write_x() {
  // 1
  x.store(true, std::memory_order_release);
}
void write_y() {
  // 2
  y.store(true, std::memory_order_release);
}
void read_x_then_y() {
  // 3
  while(!x.load(std::memory_order_acquire));
  // 4
  if (y.load(std::memory_order_acquire)) ++z;
}
void read_y_then_x() {
  // 5
  while(!y.load(std::memory_order_acquire));
  // 6
  if (x.load(std::memory_order_acquire)) ++z;
}
int main() {
  x = false;
  y = false;
  z = 0;
  std::thread a{write_x};
  std::thread b{write_y};
  std::thread c{read_x_then_y};
  std::thread d{read_y_then_x};
  a.join();
  b.join();
  c.join();
  d.join();
  assert(z.load() != 0);
}
```

- **`assert`** may fire
- given that (4) and (6) may not form a synchronizes-with relationship with (2) and (1) respectively, it is possible for the modification order to look like this:
  1. (1) - set **`x = true`**
  2. (3) - break out of loop
  3. (4) - **`y = false`**, **`z`** not incremented
  4. (2) - set **`y = true`**
  5. (5) - break out of loop
  6. (6) - does not see **`x = true`** yet, **`z`** not incremented
  - ▸ no sw relationship ⇒ no guarantee to read the value written

⋆ a single **`release`** can synchronize with any number of **`acquire`**

concurrent data structures

goal: multiple threads can access the data structure concurrently, perform operations, and each thread sees a self-consistent view of the data structure

invariants: ensure all invariants are upheld

use of mutexes over the entire data structure: prevents true concurrent access to the data it protects

- **serialization:** threads take turns accessing data

opportunity for concurrency: smaller the protected region, the fewer operations are serialized, and the greater the potential for concurrency

considerations

1. ensure no thread can see a state where the invariants are broken (during an update) by another thread
2. provide functions for complete operations, rather than operation steps to avoid race conditions
3. pay attention to exceptions to ensure invariants are not broken
4. restrict the scope of locks and avoid nested locks

fundamental object operations

includes constructors, destructors, assignment, **`swap()`**, or copy construction

- ensure data structures are not accessible during these operations
- decide if these operations are safe to call concurrently

```
// disabling copy and assignment
queue(const queue& other)=delete;
queue& operator=(const queue& other)=delete;
```

design patterns

granular locking: hold a lock for the minimum possible time needed to perform the operation

- sometimes one mutex per node/element
- allow some parts of an operation to be performed outside the lock
- protect different parts of the data structure with different mutexes
- may create synchronizes-with relationship between operations

shared access vs exclusive access:

- some operations can be performed concurrently
  - ▸ enabled using **`std::shared_mutex`**
- others might require exclusive access

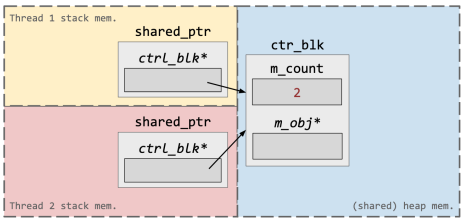
dummy nodes: improve opportunities for concurrency without affecting the operational semantics

- ensure there is at least one node to separate the node being accessed from the front and back
- adds a level of indirection to store data

possible UAF: unlock before deleting

- be careful

`std::shared_ptr`



thread-safe reference-counted smart pointer

- underlying data is not thread safe

copy: references + 1

destructed: references - 1

reaching 0: destructor automatically called on object it is pointing to

```
std::shared_ptr<T> new_data{
  std::make_shared<T>(std::move(new_value))
};
```

for full details, refer to tutorial 3

`std::unique_ptr`

thread-safe smart pointer

- can only be referenced by one owner
- ownership can be transferred

destructed: destroy the pointer and underlying data structure

```
std::unique_ptr<node> p{new node};
```

prefer `unique_ptr` over `shared_ptr`:

1. ambiguous ownership for **`shared_ptr`**
   - resources should have a single owner
2. memory leaks for **`shared_ptr`**
   - cyclic references are possible
3. performance overhead for maintaining reference counts
   - may unnecessarily hurt performance

cyclic references

beware of cyclic references created by **`shared_ptr`**. can be fixed by replacing one with a regular pointer (but need to manage the lifetime correctly)

```
struct Node {
  std::shared_ptr<Node> head;
  std::shared_ptr<Node> tail;

  Node() {
    head = new Node();
    tail = new Node();
    head.next = tail;
    tail.prev = head;
  }
}
```

queue invariants

for code, refer to lecture 5 slides

- **`back->next == nullptr`**
- **`back->data == nullptr`**
- **`front == back`** ⇒ empty list
- single element list has **`front->next == back`**
- for each node **`x`** 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`** ⇒ **`x`** is the last node

nonblocking data structures

data structures and algorithms that do not use blocking library functions

types:

1. obstruction free: all other threads are paused ⇒ any given thread will complete its operation in a bounded number of steps
2. lock free: multiple threads are operating on a data structure ⇒ after a bounded number of steps, one of them will complete its operation
3. 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 algorithms

refer to tutorial 4 for full details

when the program threads are running for a sufficiently long time, at least one of the threads makes progress

- allows individual threads to starve but guarantees system-wide throughput

⋆ all wait-free algorithms are lock-free

progress: if one or more threads get suspended, non-suspended threads must still be able to make progress by some deadline

1. if function returns
2. if effects become visible

progress conditions:

1. wait-freedom ("no one ever waits")
   - every operation completes in a finite number of steps
   - guarantees maximal progress, i.e. system-wide throughput and starvation-freedom
2. lock-freedom ("someone makes progress")
   - at least one thread makes progress
   - guarantees minimal progress, i.e. system-wide throughput but individual threads may starve
3. obstruction-freedom ("progress if no interference")
   - at any point, a single thread executed in isolation finishes in a finite number of steps
   - does not guarantee progress while two or more threads are running concurrently (livelock possible, but not deadlock)

	independent non-blocking	**dependent non-blocking**	**dependent blocking**
maximal progress	wait-free	obstruction-free	starvation-free
minimal progress	lock-free		deadlock-free

concurrent access: more than one thread must be able to access the data structure concurrently

- do not be have to be the same operations
- if one is suspended, the others must complete their operation without waiting for the suspended thread

pros:

1. enable maximum concurrency because some thread makes progress with every step
2. robustness: if a thread dies partway, nothing is lost except that thread's data

cons:

1. livelocks are possible where two threads are making changes that require the operation to restart
2. decrease overall performance but reduce individual thread waiting time
   - atomic operations used for lock-free code can be much slower
   - hardware must synchronize data between threads accessing the same atomics
     - involves memory contention and write propagation
     - cache ping-pong with multiple threads accessing the same atomic variables

guidelines:

- use `seq_cst` for prototyping
- use lock-free memory reclamation scheme
  - track how many threads are accessing an object and delete each object when no longer referenced
  - tackles use-after-free
  - recycle nodes
- watch out for ABA problem
- identify busy-wait loops and help the other thread

ABA problem

the internal data is different, but the pointer is the same, so it fails the compare and exchange comparison

using generation-counted pointers: keep a "generation" per mutation to ensure that the compare and exchange passes IF the internal changes

refer to tutorial 4 problem 2 for more details

potential serializability problems

behavior might not be intuitive because of lack of locking

- but behavior is permitted and it does not break implementation

refer to tutorial 4 problem 5 for more details

contention and cache ping-pong

one thread modifies data and change has to be propagated to the cache on the other core

- depending on memory ordering, modification may cause second core to stop and wait for the change
  - very slow
  - memory contention increases with more threads
- accessing data from the same cache line within multiple threads

false sharing: field from a struct updates and requires the entire struct to be refetched, even if the oter fields were not updated

wait-free data structures

data structures that avoid the following are wait-free

- lock-free algorithms with loops (using compare/exchange operations) can result in one thread being starved
- algorithms that can involve an unbounded number of retries because of clashes with other threads

dangers of spinlocking: very easy to write a spinlock

debugging C++ programs

for full details, refer to lecture 6 slides

ThreadSanitizer (TSan)

not complete: need to reason about every possible execution of a program which is not practical

- ⇒ even if a program has a data race, TSan might not detect it
- finding all feasible data races is at least NP-hard

task dependency graph

used to visualize and evaluate the task decomposition strategy

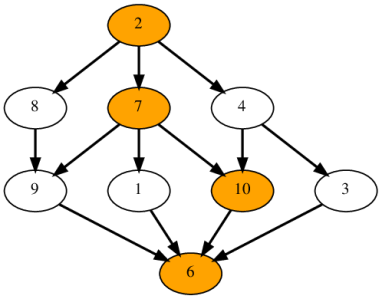
directed acyclic graph:

- nodes: each task and value is expected execution time
- edge: control dependency between task

critical path: maximum (slowest) completion time

degree of concurrency: total work / critical path length

- indication of amount of work that can be done concurrently



concurrent programming challenges

1. finding enough concurrency
2. granularity of tasks
3. coordination and synchronization

speedup

measure the benefit of parallelism

- comparison between sequential and parallel execution time

$$S_{p(n)} = \frac{T_{\text{best\_seq}(n)}}{T_{p(n)}}$$

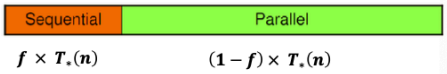
Amdahl's Law (1967)

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

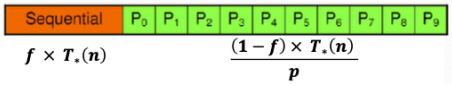
sequential fraction: $0 \leq f \leq 1$

- fixed-workload performance

- Sequential execution time:



- Parallel execution time:



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

$$= \boxed{\frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}}$$

Go

compiled and statically typed programming language announced at Google in 2009

syntactically similar to C with: memory safety, garbage collection, CSP-style concurrency

batteries included: `gc`, `gccgo`, `gollvm`

many concurrency designs: programmers can choose different levels of granularity to scale dynamically depending on the amount of parallelism supported on the program's host

- different parts of a process can be parallelised and scaled independently with proper synchronization between them
- Amdahl's law in action

Communicating Sequential Processes (CSP): structure a program by breaking it into pieces that can be executed independently (*concurrency*) and coordinate the independent executions (*communication*)

- refined to process calculus
- used to reason about program correctness

concurrency: goroutines

communication: channels and `select`

- goroutines write to `channel<-value` and read from channels `value := <-channel`
- represents the **dependencies**

goroutines

function running independently in the same address space as other goroutines

- runs on OS threads
- similar to `&` in shell
- cheaper than threads
- follow fork-join model
- represents **tasks**

goroutine scheduling: Go runtime multiplexes goroutine onto OS threads automatically `M:N` mapping

- decouples concurrency from parallelism

special class of coroutine: concurrent subroutine

- when a goroutine blocks, not other goroutines block
- preemptable: Go's runtime can suspend them

lightweight: given a few kilobytes → almost always enough

- runtime grows (and shrinks) the memory for storing the stack automatically
- CPU overhead averages about 3 cheap instructions per function call
- practical to create hundreds of thousands of goroutines in the same address space

not garbage collected: programmers should prevent goroutine leak

closures: if a variable is in a closure, it is transferred to the heap so it can be accessed after the function ends

```
func main() {
  var wg sync.WaitGroup
  for _, salutation := range []string{"hello",
"greetings", "good day"} {
    wg.Add(1)
    go func() {
      defer wg.Done()
      // salutation is closed inside the anonymous
function
      fmt.Println(salutation)
    }()
  }
  wg.Wait()
}
```

```
func main() {
  var wg sync.WaitGroup
  for _, salutation := range []string{"hello",
"greetings", "good day"} {
    wg.Add(1)
    go func(salutation string) {
      defer wg.Done()
      fmt.Println(salutation)
    }(salutation)
  }
  wg.Wait()
}
```

starting goroutines: `go <function name>` or `go func(){}()`

potential issues with shared memory: since goroutines share the same memory, we need to synchronize access to shared memory locations

- solution: *channels*

channels

reference to a place in memory where the channel resides that serves as a conduit for a stream of information, like `|` in shell

- goroutines write to `channel<-value` and read from channels `value := <-channel`
- represents the **dependencies**

⋆ *essentially* atomic, thread-free multi-producer, multi-consumer queue

no knowledge required: of the other part of the programs that work with the channel

properties:

1. typed: `make(chan int)`
2. bi-/uni- directional

```
// bi-directional channel
dataStream := make(chan interface{})

// uni-directional channels
var receiveChan <-chan interface{}
var sendChan chan<- interface{}
receiveChan = dataStream
sendChan = dataStream
```

3. blocking: depends on buffered or unbuffered
   - can cause deadlocks
4. unlimited reads from a closed channel
   - reads default value of type

ranging over channels: loop to continuously retrieve values

⋆ be careful of reading an unclosed channel forever

```
intStream := make(chan int)
for integer := range intStream {
  // ...
}
```

barriers: force all goroutines to start at the same time by reading from an unbuffered channel and closing the channel so all will start

```
begin := make(chan any)
for i := 0; i < 5; i++ {
  go func() {
    <-begin // blocks on all
  }()
}
close(begin) // unblocks all
```

unbuffered channels: writes wait for a read to occur before proceeding and reads wait for a write to occur before proceeding

```
timerChan := make(chan time.Time)
go func() {
  time.Sleep(deltaT)
  timerChan <- time.Now()
}()

// Blocks till the value is written
completedAt := <-timerChan
```

buffered channels: fixed capacity channel

- generally discouraged:
  - ▸ permutations of interleavings between producer-producer, producer-consumer, consumer-consumer increases
  - ▸ more than 1 owner to values

```
bufCh := make(chan int, 5)
```

channel behavior:

Operation	Unbuffered	Buffered	Closed	`nil`
Send	Sends if receiver available, otherwise **blocks**	Sends if not full, otherwise **blocks**	Panic!	Blocks
Receive	Receives if available, otherwise **blocks**	Receives if data available, otherwise **blocks**	Receives remaining or default value if empty	Blocks
Close	Ok	Ok	Panic!	Panic!

ownership: owner is the goroutine that instantiates, writes, and closes a channel

- useful when reasoning about program correctness
- uni-directional channels:
  - **`chan`** or **`chan<-`**: owners have a write-access view into the channel
  - **`<-chan`**: utilizers only have a read-only view into the channel

responsibilities: ownership increases safety by ensuring that we perform only operations we know will not cause any problems

Owner should	Consumer should
Initialize 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 written values and expose them via a reader channel	

`select` statement

safely brings channels together with concepts like cancellations, timeouts, waiting, and default values

- binds channels…
  - locally: within a single function or type
  - globally: at the intersection of two or more components in a system

```
select {
case <-c1:
  // ...
case <-c2:
  // ...
default:
  // optional
}
```

non-sequential cases: **`case`** statements aren't tested sequentially

- execution won't automatically fall through if none of the criteria are met
- all **`case`** are considered simultaneously
  - read: populated or closed channel
  - write: not at capacity

behavior: entire **`select`** blocks if none of the channels are ready

- unblock using **`default`**

scenarios:

1. multiple channels have something to read
   - first to match will run
   - pseudorandom uniform selection over the set of **`case`** statements
2. there are never any channels ready to write
3. no channels ready
   - use a timeout with **`time.After`**
   - or do work while waiting using **`default`**

```
var c <-chan int
select {
case <-c:
case <-time.After(1 * time.Second):
  fmt.Println("timed out")
}
```

for-select loop: allows a goroutine to make progress on work while waiting for another goroutine to report a result

```
done := make(chan interface{})
go func() {
  time.Sleep(5 * time.Second)
  close(done)
}()
workCounter := 0
loop:
for {
  select {
  case <-done:
    break loop
  default:
  }
  // Simulate work
  workCounter++
  time.Sleep(1 * time.Second)
}
fmt.Printf("%v cycles of work.\n", workCounter)
```

`sync` package

implements necessary synchronization constructs like mutexes

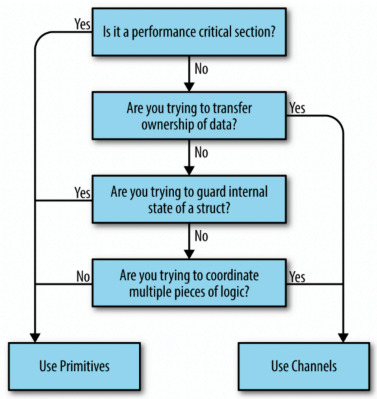
"do not communicate by sharing memory, share memory by communicating"

- used mostly in small scopes like **`struct`**

contains:

- **`WaitGroup`**: wait for a set of concurrent operations to complete
- synchronization primitives: **`Mutex`**, **`RWMutex`**, **`Cond`**, **`Once`**
- basic constructs: **`Pool`**

`sync` vs channels:



`sync.WaitGroup`

key gotchas:

- **`defer wg.Done()`** may cause ordering problems when waiting to receive values
  - to spot: think about the interactions of the last solution
  - solution: (a) move **`wg.Done()`** before pushing to channel, (b) use buffered channel (won't block anymore because buffered channels don't block when not full)
- **`wg.Add()`** should be added OUTSIDE and BEFORE the goroutine starts
  - otherwise, it may not increment in time for **`wg.Wait()`** or it may increment AFTER goroutine completes
- **`wg.Add(-1)`** is permitted, but **`wg.Done()`** on counter **`0`** panics
- **`wg.Add(1000)`** is permitted

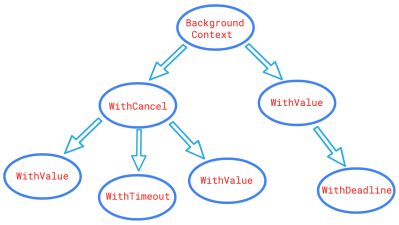
`context.Context`

refer to tutorial 5 for full details

encapsulates **`done`** channel pattern

```
for {
  select {
  case <-ctx.Done():
    // ...
  }
}

ctx, cancel := context.WithTimeout(
  context.Background(),
  time.Second,
)
```

context trees: contexts can "depend" on each other to receive signals



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

transitive closure of the union of sequenced-before and synchronized-before relations

- within a single goroutine, reads and writes must behave as if they executed in the order specified (sequenced-before)
- execution order observed by one goroutine may differ from the order perceived by another

to guarantee: 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

1. w hb r
2. any other write to v either happens before w or after r

synchronized-before

- `go` statement **synchronized-before** goroutine's execution
- exit of goroutine is not guaranteed to be **synchronized-before** any event in the program
- `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 = "hello world"
  c <- 0
}

func main() {
  go f()
  <-c
  print(a) // always prints "hello world"
}
```

- `close` of a channel is **synchronized-before** a receive that returns the default value
- `receive` from an *unbuffered channel* is **synchronized-before** the completion of the corresponding send on that channel
  - in the following, `send` happens-before `receive`, so `main()` might not see the new value of `a`

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

func f() {
  a = "hello world"
  <-c
}

func main() {
  go f()
  c <- 0
  // once the send is performed, print() can happen
  print(a) // always prints ""
}
```

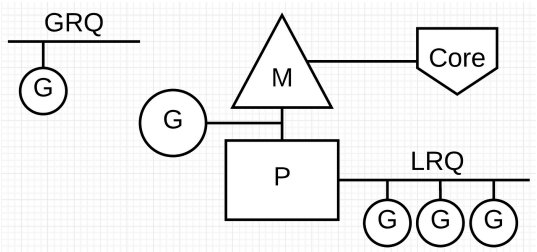
- k^{th} receive on a channel with capacity C is **synchronized-before** the $(k+C)^{\text{th}}$ send from that channel completes

Go scheduler

multiplexing goroutines onto OS threads using a work stealing strategy

naive work sharing strategies:

- fair scheduling: equally divide tasks to the number of processors
- centralized queue with tasks: locality and imbalances
- decentralized work queues



G: goroutines: placed in the global run queue (GRQ) when runnable

- contains unassigned goroutines
- moved to LRQ when assigned to OS thread

M: OS thread: runs goroutines one after another from the local run queue (LRQ) of runnable goroutines

- context-switching among multiple goroutines

P: logical processor

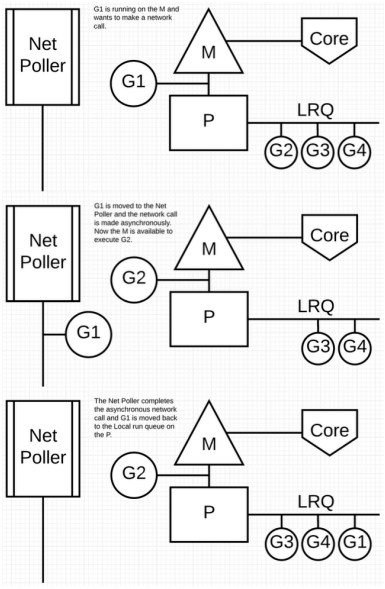
preemptive scheduler: schedule goroutines and garbage collection goroutines

goroutine states:

- waiting: goroutine is stopped and waiting for something
  - usually due to syscalls or synchronization calls (atomic and mutex operations)
- runnable: goroutine wants time on an `M` so it can execute its assigned instructions
- executing: goroutine has been placed on an `M` and is executing its instructions

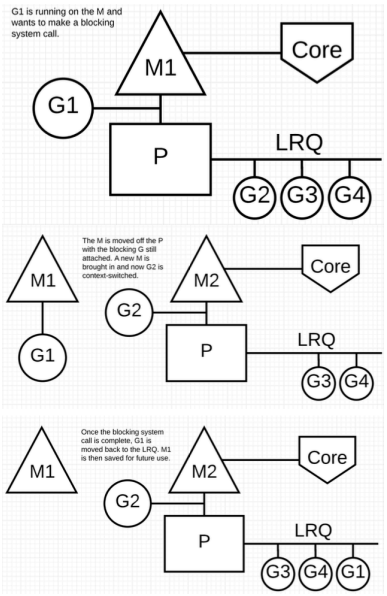
asynchronous syscalls: network-based system calls can be processed asynchronously

- network poller for handling networking operations
- no new OS threads are created ⇒ reduces scheduling load on the OS

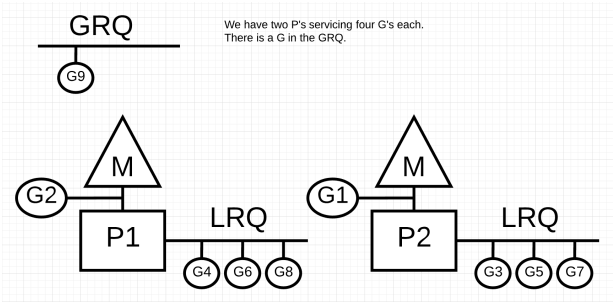


synchronous syscalls:

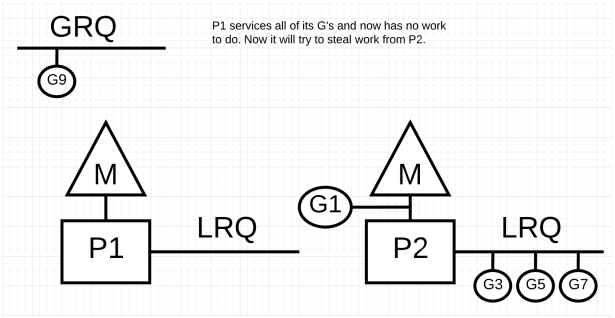
1. scheduler identifies goroutines that cause `M1` to block
2. scheduler detaches `M1` from `P` with the blocking goroutine still attached
3. scheduler brings in a new OS thread `M2` to service `P` (might be recycled)
4. when the blocking syscall `G1` finishes, `G1` moves back into the LRQ of `P`
   - `M1` is placed on the side for future use



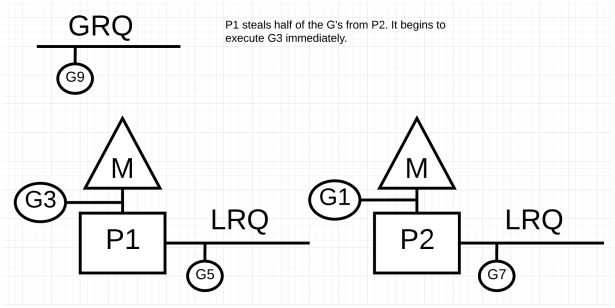
work stealing: work taken from GRQ when LRQ is empty



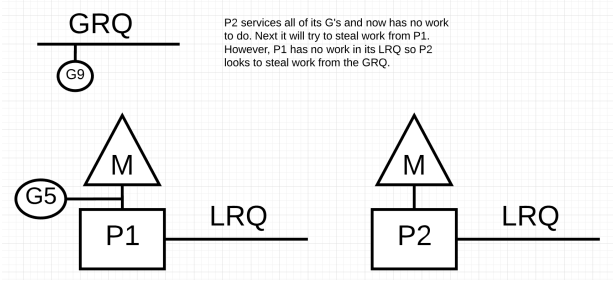
1



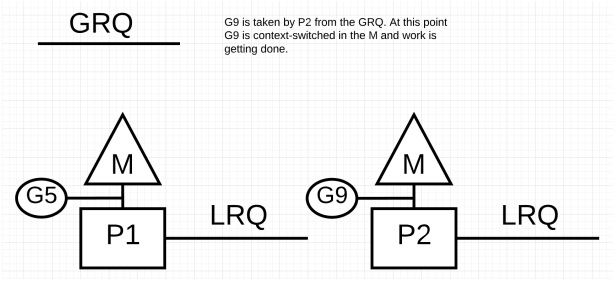
2



3



4



5

Go concurrency patterns

confinement

protecting data through data chunking and immutable data

- safe operations achieved via synchronization primitives and communication

ad-hoc confinement: data modified only from one goroutine, even if it is accessible from multiple goroutines

- needs static analysis to ensure safety

lexical confinement: restrict access to shared locations

lexical confinement through read/write-only handle: only expose the reading/writing handle of the channel

```
func main() {
  chanOwner := func() <-chan int {
    // Only write to the channel
    results := make(chan int, 5)
    go func() {
      defer close(results)
      for i := 0; i <= 5; i++ {
        results <- i
      }
    }()
    return results
  }

  consumer := func(results <-chan int) {
    // Only receive the read handle
    for result := range results {
      fmt.Printf("Received: %d\n", result)
    }
    fmt.Println("Done receiving!")
  }

  results := chanOwner()
  consumer(results)
}
```

lexical confinement through chunking: only expose a slice of the entire data

```
func main() {
  printData := func(wg *sync.WaitGroup, data []byte) {
    defer wg.Done()
    var buff bytes.Buffer
    for _, b := range data {
      fmt.Fprintf(&buff, "%c", b)
    }
    fmt.Println(buff.String())
  }

  var wg sync.WaitGroup
  wg.Add(2)
  data := []byte("golang")
  go printData(&wg, data[:3])
  go printData(&wg, data[3:])
  wg.Wait()
}
```

`for-select` loop

```
for {
  select {
  // Do some work with channels
  }
}
```

preventing goroutines from leaking

ensure goroutines terminate when

- work is completed
- unrecoverable error encountered
- told to stop

convention: if a goroutine is responsible for creating a goroutine, it is also responsible for ensuring it can stop the goroutine

leaking goroutine example:

```
func main() {
  doWork := func(strings <-chan string) <-chan any {
    completed := make(chan any)
    go func() {
      defer fmt.Println("doWork exited")
      defer close(completed)
      // forever blocks on reading nil strings
      for s := range strings {
        fmt.Println(s)
      }
    }()
    return completed
  }

  // leak memory because of nil channel
  doWork(nil) // Accumulate in memory
  fmt.Println("Done")
}
```

stopping goroutines: pass a `done` channel to the reader/writer and process using `for-select` loop

- once `done` receives a value, we just terminate any looping to avoid leaking the goroutine

error handling

gracefully handle erroneous states

responsibility: central state-goroutine

- 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 the result type, and passed along through the same lines of communication

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

func checkStatus(done <-chan any, urls ...string) <-chan
Result {
  results := make(chan Result)
  go func() {
    defer close(results)
    for _, url := range urls {
      var result Result
      resp, err := http.Get(url)
      result = Result{Error: err, Response: resp}
      select {
      case <-done:
        return
      case results <- result:
      }
    }
  }()
  return results
}

func main() {
  done := make(chan any)
  defer close(done)

  errCount := 0
  urls := []string{"a", "https://google.com", "b", "c"}
  for result := range checkStatus(done, urls...) {
    if result.Error != nil {
      fmt.Printf("error: %v\n", result.Error)
      errCount++
      if errCount >= 3 {
        fmt.Println("Too many errors, breaking!")
        break
      }
      continue
    }
    fmt.Printf("Response: %v\n", result.Response.Status)
  }
}
```

pipeline

set of data processing elements (stage) connected in series, where the output of one element is the input of the next one

stage: group of goroutines running the same function

- receive values from upstream via inbound channels
- perform some function on that data, usually producing new values
- send values downstream via outbound channels

connection: each stage is connected by channels

first stage: source or producer

last stage: sink or consumer

separation of concerns: every stage is independent

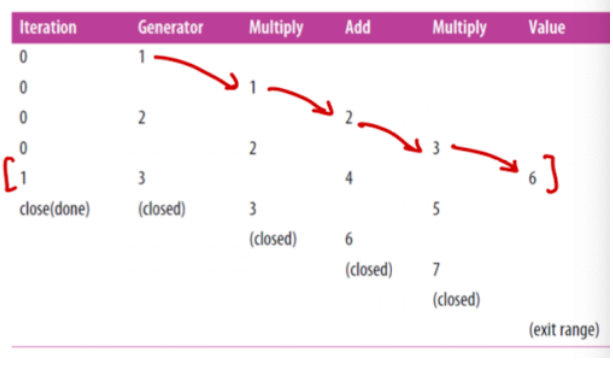
- modify each stage independently
- mix and match how stages are combined independent of modification
- process each stage concurrent to upstream or downstream stages
- fan-out or rate-limit portions of the pipeline

efficient pipeline design:

- divide the work and resources among 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

vs task pool: tweaking needed to be more efficient than a task pool

- better if there is a cap on a specific resource that is needed by all tasks in the task pool at different times (limit the stage to be executed by just 1 task)
- e.g. reading or writing to a restricted network link




```
func generator(
  done <-chan any,
  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
}

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

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

func main() {
  done := make(chan any)
  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)
  }
}
```

generic pipeline stage building: with stage scaling

```
func toPipelineStage(
  ctx context.Context,
  reqCh <-chan any,
  instances int,
  fn func(*any) *any,
) chan any {
  outCh := make(chan any)
  for i := 0; i < instances; i++ {
    go func() {
      for req := range reqCh {
        select {
        case outCh <- *fn(&req):
        case <-ctx.Done():
          return
        }
      }
    }()
  }
  return outCh
}
```

fan-out, fan-in pattern

stages in a pipeline might be slower than others and they might benefit from parallelism

fan-out: start multiple goroutines to handle input from the pipeline

- used if the stage does not rely on values that the stage had calculated before
- no fixed order or scheduling
- number of goroutines created matters
  - use `runtime.NumCPU()` to find the number of OS threads that are used to run the goroutines
  - rule of thumb: fan-out `runtime.NumCPU()` goroutines or profile the code to enhance performance

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

fan-in: combine multiple results into one channel
- involves multiplexing (joining) multiple streams of data into a single stream
- 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

⋆ no inherent ordering of events received

```
func fanIn(
  done <-chan any,
  channels ...<-chan any,
) <-chan any {
  var wg sync.WaitGroup
  multiplexedStream := make(chan any)
  multiplex := func(c <-chan any) {
    defer wg.Done()
    for i := range c {
      select {
      case <-done:
        return
      case multiplexedStream <- i:
      }
    }
  }

  wg.Add(len(channels))
  for _, c := range channels {
    go multiplex(c)
  }

  go func() {
    wg.Wait()
    close(multiplexedStream)
  }()
  return multiplexedStream
}
```

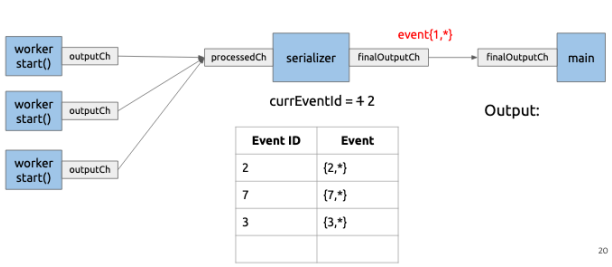
reducing values from stream: reducing fan-out results

```
var wg sync.WaitGroup
fanInCh := make(chan int)

for i := 0; i < 5; i++ {
  wg.Add(1)
  go func() {
    defer wg.Done()
    for j := 0; j < 10; j++ {
      fanInCh <- (i + 1) * (j + 1)
    }
  }()
}

// this goroutine will close the fanInCh
// only after every worker is completed
go func() {
  wg.Wait()
  close(fanInCh)
}()

sum := 0
// range fanInCh (closed) will read all values
// in the channel remaining and stop once no values
// remain
// BLOCKS the end of function until fanInCh is empty
for v := range fanInCh {
  sum += v
}
```

serializer

refer to tutorial 6 for full details

"middleman" goroutine to enforce ordering of event outputs from an input
- similar to TCP server protocol process

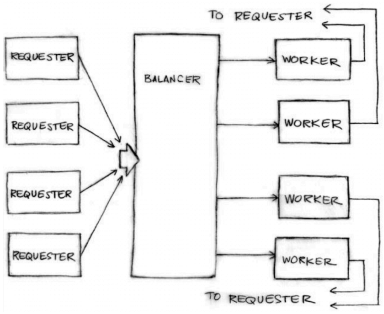


higher order channels

refer to tutorial 6 for full details

passing channels to channels
- `chan chan int`
- channels push themselves in to "reserve a spot"
  - if they do not get anything, they will not give up its spot
- useful for serializing output
- there's two orders:
  1. order event received (push channel)
  2. order response received (block till first order received pops)

load balancer



requests:

```
type Request struct {
  fn func() int
  c chan int
}
```

requesters: send `Request` to the balancer

```
func requester(work chan<- Request) {
  c := make(chan int)
  for {
    Sleep(rand.Int63n(nWorker * 2 * Second))
    work <- Request{workFn, c}
    result := <-c
    furtherProcess(result)
  }
}
```

workers: receives requests and contains load tracking information

```
type Worker struct {
  requests chan Request
  pending int
  index int
}

func (w *Worker) work(done chan *Worker) {
  for {
    req := <-w.requests
    req.c <- req.fn()
    done <- w
  }
}
```

balancer: contains a pool of workers and result channel for requesters

```
type Pool []*Worker

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

type Balancer struct {
  pool Pool
  done chan *Worker
}

func (b *Balancer) dispatch(req Request) {
  w := heap.Pop(&b.pool).(*Worker)
  w.requests <- req
  w.pending++
  heap.Push(&b.pool, w)
}
```

```
func (b *Balancer) completed(w *Worker) {
  w.pending--
  heap.Remove(&b.pool, w.index)
  heap.Push(&b.pool, w)
}

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

improving the load balancer:
1. quantify "busy-ness" of a worker
2. create more workers dynamically

increasing concurrency:
- perform work stealing when worker is completed

classical synchronization problems

Problem	CS Problem
barrier	wait until threads/processes reach a specific point in execution
producer-consumer	model interactions between a processor and devices that interact through FIFO channels
readers-writers	model access to shared memory
dining philosophers	allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner
barbershop	coordinating the execution of a processor
FIFO semaphore	needed to avoid starvation and increase fairness in the system
H2O	allocation of specific resource to a process
cigarette smokers	the agent represents an operating system that allocates resources, and the smokers represent applications that need resources

producer consumer

processes share a buffer (bounded or unbounded)

producer: produce items to insert into the buffer
- only when the buffer is not full ($< K$ items)

consumer: remove items from the buffer
- only when the buffer is not empty (> 0 items)

Producer

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

Consumer

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

  Consume item;
}
```

reader writer

processes share a data structure `D`

reader: retrieves information from `D`
- can access with other readers

writer: modifies information in `D`
- must have exclusive access to `D`

Reader

```
mutex.wait()
readers += 1
if readers == 1:
  # first reader locks the
room
  roomEmpty.wait()
mutex.signal()

# critical section for
readers

mutex.wait()
readers -= 1
if readers == 0:
  # last reader unlocks the
room
  roomEmpty.signal()
mutex.signal()
```

Writer

```
roomEmpty.wait()

# critical section for
writers

roomEmpty.signal()
```

lightswitch: abstracting locking and unlocking behavior through semaphores and mutexes

```
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()
```

starving writers: continuous readers means that writers don't get to write
- introduce a turnstile semaphore `turnstile = Semaphore(1)`
- writers turn the turnstile which prevents readers from reading
  - but this can also cause starvation of readers

Reader

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

Writer

```
turnstile.wait()
roomEmpty.wait()

# critical section for
writers

turnstile.signal()
roomEmpty.signal()
```

reader-writers lock: implemented in different languages
- C++17 onwards: `shared_mutex`
- Go: `RWLock`

```
mutable std::shared_mutex m;
unsigned int value = 0;

unsigned int get() const {
  std::shared_lock lock{m};
  return value;
}

unsigned int increment() {
  std::unique_lock lock{m};
  return ++value;
}

void reset() {
  std::unique_lock lock{m};
  value = 0;
}
```

barrier

refer to the "Little Book of Semaphores" for full details

any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier

single use barrier (latch): starts in the raised state and cannot be re-raised once it is in the lowered state
- `std::latch`

reusable barriers: once the arriving threads are unblocked from a barrier's phase's synchronization point, the same barrier can be reused
- combining tree barrier: hierarchical way of implementing barrier
  - resolve the scalability by avoiding the case that all threads are spinning at the same location
- `std::brrier`
- `WaitGroup#wait`

usage: appears in many collective routines as part of directive-based parallel languages
- parallel `for` loop in OpenMP
- collective communication in `MPI`

```
std::barrier sync_point{4, on_completion};

void foo() {
```

```
  sync_point.arrive_and_wait();
}
```

C++ implementation:

- note that just using a single **`turnstile`** may result in threads going ahead of others by one lap
  - a thread can pass through the second mutex and loop around and pass through the first mutex and turnstile
  - this causes it to open the barrier early
- fix: use two turnstiles: first is locked, second is open
  - when all threads arrive at the first, we lock the second and unlock the first
  - when all threads arrive at the second, we relock the first, making it safe for threads to loop around to the beginning, and then we open the second

```
mutex.wait()
count += 1
if count == n:
  turnstile.signal(n)
mutex.signal()

turnstile.wait()

# critical section

mutex.wait()
count -= 1
if count == 0:
  turnstile2.signal(n)
mutex.signal()

turnstile2.wait()
```

Preloaded barrier

Go implementation: use 2 **`sync.WaitGroup`** as a turnstile

```
type Barrier struct {
  wg  sync.WaitGroup
  wg2 sync.WaitGroup
}

func (b *Barrier) Init(expected int) {
  b.wg.Add(expected)
  b.wg2.Add(expected)
}

func (b *Barrier) Wait() {
  b.wg.Done()
  b.wg.Wait()
  b.wg.Add(1)

  // critical section

  bg.wg2.Done()
  bg.wg2.Wait()
  bg.wg2.Add(1)
}
```

dining philosophers

refer to the "Little Book of Semaphores" for full details

models the problem of allocating limited resources to a group of processes in a deadlock-free and starvation-free manner

naive implementation:

- deadlock occurs if all philosophers simultaneously takes up the left chopstick and none can take the right chopstick
- fix: make the philosopher put down the left chopstick if the right chopstick cannot be acquired
  - may result in livelock

```
N = 5
LEFT = i
RIGHT = ((i + 1) % N)

# for philosopher i
while (True):
  Think()
  takeChpStick(LEFT)
  takeChpStick(RIGHT)

  Eat()

  putChpStick(LEFT)
  putChpStick(RIGHT)
```

C++ implementation: use **`std::scoped_lock`** or **`lock()`** with deadlock avoidance algorithm

```
struct DiningTable {
  using ChpStick = std::mutex;

  ChpStick chpSticks[N];

  ChpStick& get_left_chpStick(size_t pid) {
    return  chpSticks[pid];
  }

  ChpStick& get_right_chpStick(size_t pid) {
    return chpSticks[(pid + 1) % N];
  }

  void eat(
    size_t pid,
    void (*eat_callback)(size_t pid)
  ) {
    std::scoped_lock l{
      get_left_chpStick(pid),
      get_right_chpStick(pid)
    };
    eat_callback(pid);
  }
}
```

Go implementation: use odd-even ring communication to avoid deadlock

- if **`pid`** is even, pick the left chopstick first, then right chopstick
- if **`pid`** is odd, pick the right chopstick first, then left chopstick

```
type ChpStick struct{}

type DiningTable struct {
  numPhilosophers int
  chpStickChs     []chan ChapStick
}

func (t *DiningTable) 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 *DiningTable) leftChopStickCh(pid int) chan
ChpStick {
  return t.chpStickChs[pid]
}

func (t *DiningTable) rightChopStickCh(pid int) chan
ChpStick {
  return t.chpStickChs[(pid + 1) % t.numPhilosophers]
}

func (t *DiningTable) evenChpStickCh(pid int) chan
ChpStick {
  if pid % 2 == 0 {
    return t.leftChopStickCh(pid)
  } else {
    return t.rightChopStickCh(pid)
  }
}

func (t *DiningTable) oddChpStickCh(pid int) chan ChpStick
{
  if pid % 2 == 0 {
    return t.rightChopStickCh(pid)
  } else {
    return t.leftChopStickCh(pid)
  }
}

func (t *DiningTable) Eat(
  pid int,
  eat_callback func(pid int),
) {
  evenChpStickCh := t.evenChpStickCh(pid)
  oddChpStickCh := t.oddChpStickCh(pid)

  <-evenChpStickCh
  <-oddChpStickCh

  eat_callback(pid)

  evenChpStickCh <- ChpStick{}
  oddChpStickCh <- ChpStick{}
}
```

Go implementation: use swap order of chopsticks

```
type ChpStick struct{}

type DiningTable struct {
  numPhilosophers int
  chpStickChs     []chan ChapStick
}

func (t *DiningTable) 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:
      // Could not get oddChpStickCh, swap
    }
    evenChpStickCh <- ChpStick{}

    <-oddChpStickCh
    select {
    case <-evenChpStickCh:
      break breakLock
    default:
      // Could not get evenChpStickCh, swap
    }
    oddChpStickCh <- ChpStick{}
  }

  eat_callback(pid)

  evenChpStickCh <- ChpStick{}
  oddChpStickCh <- ChpStick{}
}
```

Tanenbaum solution: for each philosopher there is a state variable that indicates whether the philosopher is thinking, eating, or waiting to eat

- semaphore indicates whether the philosopher can eat
- philosopher can eat when
  1. philosopher finds available chopsticks and proceeds
  2. one of the neighbors are eating and the philosopher waits, so once the neighbor finishes, it will try to pick up chopsticks again

```
N = 5
LEFT = i
RIGHT = ((i + 1) % N)

class State(Enum):
  THINKING
  HUNGRY
  EATING

state: List[State] = [State.THINKING] * N
mutex: Semaphore = 1
s: List[Semaphore] = 1

def safeToEat(i):
  if state[i] == State.HUNGRY and \
    state[LEFT] != State.EATING and \
    state[RIGHT] != State.EATING:
    state[i] = EATING
    signal(s[i])

def takeChpSticks(i):
  wait(mutex)
  state[i] = State.HUNGRY
  safeToEat(i)
  signal(mutex)
  wait(s[i])

def putChpSticks(i):
  wait(mutex)
  state[i] = THINKING
  safeToEat(LEFT)
  safeToEat(RIGHT)
  signal(mutex)

def philosopher(i):
  while (True):
    Think()
    takeChpSticks(i)
    Eat()
    putChpSticks(i)
```

other implementations:

1. limited eater: number of philosophers is one less than total number of seats ⇒ never deadlock
2. using footman semaphore: limit number of eaters
   - starvation is possible since mutexes and semaphores are not fair in C++
3. using FIFO semaphores

barbershop

barbershop consists of a waiting room with `n` chairs and the barber chair

- if no customers are to be served, the barber goes to sleep
- if the barber is busy, but chairs are available, then customer sits in one of the free chairs
- if the barber is asleep, the customer wakes up the barber
- if a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop

Initialization

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

Customer

```
wait(mutex)
if customers == n:
  signal(mutex)
  exit()
customers++
signal(mutex)

signal(customer)
wait(barber)
getHairCut()
signal(customerDone)
wait(barberDone)

wait(mutex)
customers--
signal(mutex)
```

Barber

```
while (true):
  wait(customer)
  signal(barber)
  cutHair()
// sw customer signal
  wait(customerDone)
// sw customer wait
  signal(barberDone)
```

variations:

1. FIFO barbershop
2. multiple barbers

H2O

refer to tutorial 7 for full details

hydrogen and oxygen atoms arrive at the factor, 2H should be bonded with 1O

- bonding can only occur when all 3 molecules arrive

C++ implementation:

- be aware of unrestricted barriers (i.e. no ordering)
- use barriers as synchronization points
- avoid using latches if the use case is more than once

```
struct WaterFactory {
  std::counting_semaphore oSem{1}, hSem{2};
  std::barrier<> barrier;
  WaterFactory() : barrier {3} {}

  void oxygen(void (*bond)()) {
    oSem.acquire();
    barrier.arrive_and_wait();
    bond();
    oSem.release();
  }

  void hydrogen(void (*bond)()) {
    hSem.acquire();
    barrier.arrive_and_wait();
    bond();
    hSem.release();
  }
}
```

message passing with Go — daemon goroutine:
coordinates all atoms with a 2-phase commit

1. (precommit) receive arrival requests from 2H and 1O
2. (commit) tell the 3 atosm to proceed and begin bonding
3. (postcommit) wait for all 3 atoms to finish bonding

```
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)
      h1 := <-wfd.precomH
      h2 := <-wfd.precomH
      o := <-wfd.precomO

      // Step 2: (Commit)
      h1 <- struct{}{}
      h2 <- struct{}{}
      o <- struct{}{}
```

```
      // Step 3: (Postcommit)
      <-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)
}
```

downsides of daemon:

- memory leaks: even if all other references to the water factory is dropped, the water factory is not cleaned up since the daemon still holds a reference
- goroutine leak: daemon goroutine does not go away and leaks memory
  - ▸ can be fixed with **`Context`**

message passing with Go — oxygen leader: use oxygen as the leader to coordinate (similar to daemon)

- use oxygen because only 1 is required

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

func NewFactoryWithLeader() WaterFactoryWithLeader {
    wf := WaterFactoryWithLeader{
        oxygenMutex: make(chan struct{}, 1),
        precomH:     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)
    h1 := <-wf.precomH
    h2 := <-wf.precomH

    // Step 3: (Commit)
    h1 <- struct{}{}
    h2 <- struct{}{}

    // Step 4: Bond
    bond()

    // Step 5: (Postcommit)
    <-h1
    <-h2

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

FIFO semaphore

refer to tutorial 7 for full details

provide starvation-freedom for accessing a semaphore

ticket queue: similar to "Serializer" pattern where we track "arrivals" of each **`acquire()`** call, and ensure that order of handling release matches order of arrival

- even if **`release()`** called out of sequence, the **`now_serving`** enforces some order

```
struct FIFOSemaphore {
  std::atomic<std::ptrdiff_t> next_ticket{1};
  std::atomic<std::ptrdiff_t> now_serving;

  FIFOSemaphore(std::ptrdiff_t initial_count) :
    now_serving{initial_count} {}

  void acquire() {
    auto my_ticket = next_ticket.fetch_all(
      1,
      std::memory_order_relaxed
    );
    while (
      now_serving.load(std::memory_order_acquire) <
my_ticket
    ) {};
  }

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

ticket queue with cond var: use conditional variable instead of busy waiting

- make sure to **`notify_all()`** to avoid starvation

```
struct FIFOSemaphore {
  std::mutex m;
  std::condition_variable cv;
  std::atomic<std::ptrdiff_t> next_ticket{1};
  std::atomic<std::ptrdiff_t> now_serving;

  FIFOSemaphore(std::ptrdiff_t initial_count) :
    now_serving{initial_count} {}

  void acquire() {
    auto my_ticket = next_ticket.fetch_all(
      1,
      std::memory_order_relaxed
    );
    std::unique_lock l{m};
    cv.wait(
      l,
      []{
        return now_serving.load(
          std::memory_order_acquire
        ) >= my_ticket;
      }
    );
  }

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

queue of semaphores: use a **`std::queue`** to fix the order of semaphores to unlock

- use **`std::binary_semaphore`** over **`std::mutex`** to allow signalling outside of the thread that acquires it
  - ▸ **`mutex`** is preferred for exclusive access
- **`count`** can be positive so that acquirers can decrement it and skip adding themselves to the queue

```
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
  }
};
```

Go buffered channel: default behavior

daemon goroutine: acquirers send a channel to daemon and block

- daemon keeps them waiting or unblocks them by closing their channel
- similar to "queue of semaphores" except it uses a goroutine to manage the queue

```
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{}{}
}
```

Rust

initiated with two goals in mind: safety in systems programming and painless concurrency

provides:

- strong safety guarantees: no seg-faults, no data races, expressive type system
- uncompromised performance: no garbage collector, no runtime, same level of performance as C/C++

unsafe C++: dangling pointers, aliased pointers, mutations on pointers

tools:

- Rustup: install Rust tools
- Rustc: Rust compiler
- Cargo: package manager
- package: one or more crates that provide a set of functionality
- crate: binary or library
- module: organize code within a crate into groups

ownership and borrowing

prevents aliasing (more than one pointer to the same memory) and mutations (changing a pointer)

aliasing + mutation: changing pointers that point to the same chunk of memory

default ownership

```
fn main() {
  let mut book = Vec::new();
  book.push(...);
  book.push(...);
  publish(book);
  // this next call to publish will fail
  // publish(book);
}

fn publish(book: Vec<String>) {
  // ...
}
```

- initially, **`book`** is owned by the **`main()`**
- after **`publish(book)`** is called without **`&`**, ownership is transferred to **`publish()`**
  - scope of **`book`** ends at the end of **`publish()`**, not **`main()`**

ownership disallows aliasing: only the owner of the memory can modify it

ownership in Rust: not like **`copy`** constructor in C++, more like **`move`**

- enforced at compilation time and no ownership is retained
- **`copy`** constructor in C++ same as **`clone()`** in Rust ⇒ deep copy

immutable borrow (shared borrow)

allows aliasing but no mutation

- freezes the whole container vector
  - individual elements are not "unfrozen", even if **`&mut a[0]`** is done

```
fn main() {
  let mut book = Vec::new();
  book.push(...);
  book.push(...);
  publish(&book);
  // this next call to publish will succeed
  publish(&book);
}

fn publish(book: &Vec<String>) {
  // ...
}
```

- **`&Vec<String>`** is a reference to a vector
- **`&book`** creates an immutable reference

⋆ no mutation is allowed while a shared borrow exists

- any number of shared borrows are allowed

```
let mut book = Vec::new();
book.push(...); // success: `book` is mutable

{
  let r = &book; // shared borrow of `book`
  book.push(...); // compilation error:
                  //   cannot mutate while shared
  r.push(...); // compilation error:
               //    cannot mutate while shared
} // shared borrow ends

book.push(...); // success: `book` is mutable again
```

mutable borrow

ownership is retained with original owner

- allows mutation but no aliasing

```
fn main() {
  let mut book = Vec::new();
  book.push(...);
  book.push(...);
  publish(&mut book);
  // this next call to publish will succeed
  publish(&mut book);
  // main() can continue to modify `book`
}

fn publish(book: &mut Vec<String>) {
  // ...
}
```

- **`&mut Vec<String>`** is a mutable reference to a vector
- **`&mut book`** creates a mutable borrow

⋆ no aliasing is allowed while a mutable borrow exists (`&T` not allowed)
- only the mutable borrow can be used to mutate the memory
- only one at a time

```
let mut book = Vec::new();
book.push(...); // success: `book` is mutable

{
  let r = &mut book; // mutable borrow of `book`
  book.len; // compilation error:
            //   cannot access while mutable borrow exists
  r.push(...); // success: reference can be mutated
} // mutable borrow ends

book.push(...); // success: `book` is mutable again
```

non-lexical lifetimes

```
fn main() {
  let mut scores = vec![1, 2, 3];
  let score = &scores[0]; // creates a shared reference to
`scores`
  scores.push(4); // mutation still allowed
}
```

- above will compile because Rust relies on both its type system and live-variable analysis

live-variable analysis: data-flow analysis to calculate the variables that are live at each point in the program
- live: holds a value that may be needed in the future

memory safety

borrow checker statically prevents aliasing and mutation during compile time

ownership: prevents double-free as only the owner frees

borrowing: prevents use-after-free

no segfaults: avoiding using borrow checker

Type	Ownership	Alias?	Mutate?	Count
`T`	Owned		Yes	1
`&T`	Shared reference	Yes		Many
`&mut T`	Mutable reference		Yes	1

no data races: sharing + mutation at the same time are prevented in Rust

no iterator invalidation: when the iterator points to something invalid but still permitted to operate

no dangling references: each reference has a lifetime

lifetimes

proves to the compiler that we always have valid references in our program

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

fn main() {
  let string1 = String::from("abcd"); // heap allocated
String
  let string2 = "xyz"; // possibly stack allocated &str

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

- above does not compile because the returned reference `&str` does not specify `x` or `y`

lifetime annotations: specify that the lifetime of the returned value is the same as the smaller of the lifetimes of `x` and `y`

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

multiple lifetimes: specify that the return value follows the lifetime of a specific argument

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
  if x.len() > y.len() {
    println!("X is longer!")
  } else {
    println!("Y is longer!")
  }
  x
}
```

elision rules: omission of something that could be explicitly specified but intentionally left out
1. each elided lifetime in input position becomes a distinct lifetime parameter
   - if there is exactly one input lifetime position (elided or not), that lifetime is assigned to all elided output lifetimes
2. if there are multiple input lifetime positions, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to *all* elided output lifetimes
3. otherwise, it is an error to elide an output lifetime

zero-cost abstraction

code generated that you cannot write better by hand
- abstraction layers that disappear at compile-time

iterators API: in release mode, machine code is indistinguishable from hand-written implementation
- includes compiler optimization for cache locality

unsafe Rust

disable compiler checks to avoid compilation errors for *proven safe code* that the compiler cannot pick up on
- equivalent of writing C++

```
impl <T: ?Sized> Deref for MutexGuard<'_, T> {
  type Target = T;

  fn deref(&self) -> &T {
    unsafe { &*self.lock.data.get() }
  }
}
```

traits

like an interface that can implemented for a given type which may include methods

```
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;
  }
}
```

marker traits:
- `Send`: transferred across thread boundaries
- `Sync`: safe to share references between threads
  - type `T` is `Sync` iff `&T` is `Send`
- `Copy`: safe to `memcpy` (for built-in types, not for `String`)

static objects

global variables (static) whose lifetime exists across ALL threads
- requires interior mutability to mutate them as implicitly shared
- must also implement `Sync` thread to be thread safe

```
use std::thread;
use std::sync::atomics::{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));
}
```

threading

⋆ not required to be joined
- ⇒ variables that the thread borrows live as long as the process
  - thread may outlive the process ⇒ violating ownership

satisfiability: programs that satisfies ownership ⇒ not buggy

- non-buggy program ⇏ satisfy ownership

create a thread: `thread::spawn(|| {});`

- capture everything as a borrow

```
// `loc` is a JoinHandle
// if `loc` dropped, spawned thread detached
let loc = thread::spawn(|| {
  "world"
});
println!("Hello, {}!", loc.join().unwrap());
```

transfer variables: `move` converts any variables captured by reference or mutable reference to variables captured by value

- closure takes ownership of the value it uses from the environment
- capture type inferred by Rust

```
let mut dst = Vec::new();
thread::spawn(move || {
  dst.push(3);
});
dst.push(3); // error: use after move
```

- without `move`, error because of possible use-after-free
  - variable could be destroyed before the thread finishes

reference counted type (Rc)

shared pointer

```
let v = Rc::new(vec![1, 2, 3]);
let v2 = v.clone(); // increments the reference count
thread::spawn(move || {
  println!("{}", v.len()); // error
});
another_fn(v2.clone());
```

- `Rc<T>` cannot be sent across threads because it is not atomically managed
  - missing `Send` trait

atomic reference counted (Arc)

allows only shared immutable references

- `Arc<Mutex<i32>>`: atomic reference counted mutex
- `Mutex<Arc<i32>>`: mutex atomic reference counted (WRONG)

```
let v = Arc::new(vec![1, 2]);
let v2 = v.clone();
thread::spawn(move || {
  println!("{}", v.len());
});
another_fn(&v2);
```

- `Arc` is the owner of the memory
- `move` references `v` but all that does is increase the reference count

synchronization

supports both shared memory access and message passing (combines C++ and Go)

shared memory: mutex and atomics

message passing: channels using multi-producer, single-consumer FIFO queues

mutex

lock returns a guard, acting as a proxy to access data

```
let mut mutex = Mutex::new(0);
```

```
fn sync inc(counter: &Mutex<i32>) {
  let mut data MutexGuard<i32> = counter.lock();
  *data += 1;
}
```

interior mutability: `std::Mutex` uses unsafe Rust to get a mutable reference from `UnsafeCell`

- locked mutex allows only 1 thread to access the data

```
// Unsafe data
pub struct UnsafeCell<T: ?Sized> {
  value: T,
}

// Actual mutex to protect data
pub struct Mutex<T: ?Sized> {
  inner: sys::Mutex,
  poison: poison::Flag,
  data: UnsafeCell<T>,
}
```

lock poisoning: if a thread has panicked while holding the mutex, it is likely to leave data in an inconsistent state

- other threads that try to acquire the mutex should be aware of this possibility
- use `unwrap` to allow panics to propagate across threads

atomics

same memory model as C++

```
let number = AtomicUsize::new(10); // not mutable
// returns the previous value
let prev = number.fetch_add(1, SeqCst);
assert_eq!(prev, 10);
// returns the previous value
let prev = number.swap(2, SeqCst);
assert_eq!(prev, 11);
assert_eq!(number.load(SeqCst), 2);
```

- `number` is not mutable so this must be mutated through a shared reference

multi-producer, single-consumer (mpsc) FIFO queue

channel with a reading and writing reference

```
let (wx, rx) = mpsc::channel();
let wx2 = wx.clone(); // cannot clone rx
thread::spawn(move || wx.send(5));
thread::spawn(move || wx2.send(4));

// prints 4 and 5 in unspecified order
println!("{:?}", rx.recv);
println!("{:?}", rx.recv);
```

concurrency library: Crossbeam

ability to create scoped threads

- since added to `std`

scoped threads

scope: container to put threads in

- cannot borrow variables mutably into two threads in the same scope
  - by default, this is true
- uses message passing using `mpsc`
  - with exponential backoff
- all spawned threads in scope joins after scope ends

```
fn main() {
  let v = vec![1, 2, 3];
  println!("main thread has id {}", thread_id::get());

  std::thread::scope(|scope| {
    scope.spawn(|inner_scope| {
      println!("Here's a vector: {:?}", v);
      println!(
        "Now in thread with id {}",
        thread_id::get()
      );
    });
  }).unwrap(); // automatically joined

  println!("Vector v is back: {:?}", v);
}
```

fixing multiple mutable borrows: occurs when scope container spawns multiple threads that access the same variable

for full details, refer to tutorial 8

- replace variable with `Mutex`

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

multi-producer, multi-consumer

bounded channel, similar to buffered channel in Go

```
fn main() {
  let (wx, rx) = bounded(CHANNEL_CAPACITY);

  let mut threads = vec![];
  for _i in 0..NUM_THREADS {
    let wx = wx.clone();
    threads.push(
      thread::spawn(move || {
        for _k in 0..ITEMS_PER_THREAD {
          let produced = produce_item();
          wx.send(produced).unwrap();
        }
      });
    )
  }

  for _j in 0..NUM_THREADS {
    let rx = rx.clone();
    threads.push(
```

```
      thread::spawn(move || {
        for _k in 0..ITEMS_PER_THREAD {
          let consumed = rx.recv().unwrap();
          consume(consumed);
        }
      });
    )
  }

  for t in threads {
    let _ = t.join();
  }
  println!("Done!");
}
```

exponential backoff

retry in exponential delays

- avoid overloading a resource too frequently

```
use crossbeam_utils::Backoff;
use std::sync::atomic::AtomicUsize;
use std::sync::atomic::Ordering::SeqCst;

fn fetch_mul(a: &AtomicUsize, b: usize) -> usize {
  let backoff = Backoff::new();
  loop {
    let val = a.load(SeqCst);
    // lock-free looping
    if a.compare_and_swap(
      val,
      val.wrapping_mul(b),
      SeqCst
    ) == val {
      return val;
    }
    backoff.spin();
  }
}

fn spin_wait(ready: &AtomicBool) {
  let backoff = Backoff::new();
  while !ready.load(SeqCst) {
    // another thread can run instead
    backoff.snooze();
  }
}
```

concurrency library: Rayon

data parallelism library to parallelize some spots without full/major rewrite

- similar functionality to OpenMP except OpenMP uses compiler directives

rationale: reasonably common that computationally-intensive parts of the program happen in a loop

- parallelize the loop

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

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.");
  }
}
```

- `n` is a reference to `i64`: `&i64`
- `*n` dereferences it

asynchronous programming

thread overhead

threads incur both a context switching cost and memory overhead

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 with multiple threads, each handling a connection
- **solution:** lightweight threads (green threads)
  - need to managed by runtime (like Go)

memory overhead: each thread has its own stack space that needs to be managed by the OS

non-blocking I/O

enables concurrency with one thread

blocking I/O: `read()` syscall blocks if there is more data to be read but not available

- thread gets pulled off the CPU and cannot do anything else in the meantime

non-blocking I/O: `read()` returns special error value instead of blocking

- if client hasn't sent anything, thread can do other useful work

`epoll`: kernel-provided mechanism that notifies which file descriptors are ready for I/O

- scenario: server having conversations with multiple clients at the same time (multiple file descriptors)
- `epoll` notifies the server when a client says something
- `read()` from each of the available file descriptors continuing the conversation
- used in `nginx` as one of the connection processing methods

```
while (true) {
  "hey epoll, what's ready for reading?"
  epoll => [7, 12, 15]
  "thanks epoll"
  read(7)
  read(12)
  read(15)
}
```

past method:

each connection would be its thread, but if this connection is blocked on I/O, it will be suspended entirely, when it could have been used for other tasks

state management challenges: managing one connection in each thread is easy because each conversation is an independent train of thought

- introducing one thread to maintain all connections introduce additional overhead

futures

represents a value that will exist sometime in the future, but calculation has not happened yet

- allow us to keep track of in-progress operations along with associated state, in one package

runtime: event loop to continuously poll the future if it's ready

- runs behavior when it can be run
- user-space scheduler

futures in Rust: Futures.rs

zero-cost abstraction for futures in Rust

- code in the binary has no allocation and no runtime overhead in comparison to writing it by hand (non-assembly code)
- made possible with the `Future` trait

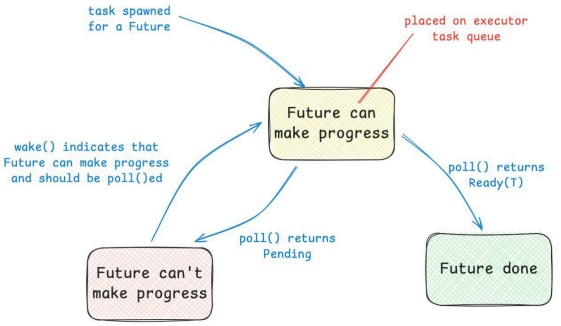
`Future` trait: each executor thread calls `poll()` on the future to start it off

- runs code till it can no longer progress

```
trait Future {
  fn poll(&mut self, cx: &mut Context) ->
Poll<Self::Output>;
}

enum Poll<T> {
  Ready(T),
  Pending,
}
```

- when future completes, return `Poll::Ready(T)`
- when future is waiting for some event, return `Poll::Pending`
  - allows the single thread to work on another future
- on `poll()`, `Context` structure passed in
  - includes `wake()` function that is called when future can make progress again
    - implemented internally using syscalls

- after **`wake()`**, executor use **`Context`** to see which future can be polled to make new progress



executors: 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, goes to sleep until one or more futures call **`wak()`**
- popular executors: **`Tokio`**
- maybe single-threaded or multi-threaded
- running on multi-core machines: futures can truly run in parallel
  - need to protect shared data using synchronization primitives

non-blocking: otherwise, if code within a future causes the thread to sleep, the executor running the code is going to sleep
- asynchronous code needs to use non-blocking versions of everything, including mutexes, and blocking syscalls
- non-blocking implementations provided by executor runtimes like **`Tokio`**

composition: combined using combinators (sequential or concurrent)

```
let future = placeOnStove(meat)
  .then(|meat| cookOneSide(meat))
  .then(|meat| flip(meat))
  .then(|meat| cookOneSide(meat))
```

- not the best ergonomics, so there is syntactic sugar

```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error> {
  let message = loadMessage(email_id).await?;
  let recipient = get_recipient(recipient_id).await?;
  recipient.verifyHasSpace(&message)?;

  recipient.addToInbox(message).await
}
```

async/await: syntactic sugar around Future code to simplify code
- **`async`** function returns a Future
- **`.await`** waits for a future and gets its value
  - can only be called in an **`async`** function or block
  - Tokio executor can choose to run each **`.await`** task on a different thread
- Rust compiler transforms this into Future with a **`poll()`** method

under the hood

Rust will transform the **`async`** function into one with the following signature

```
// This function:
async fn foo(x: &u8) -> u8 { *x }

// Is equivalent to this function:
fn foo_expanded<'a>(x: &'a u8)
    -> impl Future<Output = u8> + 'a {
    async move { *x }
}
```

refer to tutorial 9 bonus slides for more details

TCP server:

without

```
use std::io::{Read, Write};
use std::net::TcpListener;
use std::thread;

fn main() {
  let listener =
    TcpListener::bind(
      "127.0.0.1:8080"
    ).unwrap();
  loop {
    let (mut socket, _) =
      listener.accept().unwrap();
    thread::spawn(move || {
      let mut buf = [0; 1024];
      let n = socket
        .read(&mut buf).unwrap();
      socket.write_all(&buf[0..n])
        .unwrap();
    });
  }
}
```

with

```
use tokio::io::{AsyncReadExt,
AsyncWriteExt};
use tokio::net::TcpListener;
use std::thread;

#[tokio::main]
async fn main() {
  let listener =
    TcpListener::bind(
      "127.0.0.1:8080"
    ).unwrap();
  loop {
    let (mut socket, _) =
      listener.accept()
        .await.unwrap();
    tokio::spawn(async move || {
      let mut buf = [0; 1024];
      let n = socket
        .read(&mut buf)
        .await
        .unwrap();
      socket.write_all(&buf[0..n])
        .await
        .unwrap();
    });
  }
}
```

- **`#[tokio::main]`** submits the **`main()`** Future to the executor

state management: can be managed through an **`enum`**

```
enum AddToInboxState {
  NotYetStarted { email_id: u64, recipient_id: u64 },
  WaitingLoadMesssage {
    recipient_id: u64, state: LoadMessageFuture },
  WaitingGetRecipient {
    message: Message, state: GetRecipientFuture },
  WaitingAddToInbox { state: AddToInboxFuture },
  Completed { result: Result<(), Error> },
}

// within a Future
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,
      }
    }
    // ...
  }
}
```

no stack: async functions don't have a stack
- stackless coroutines
- executor thread still has a stack but it isn't used to store state when switching between async tasks
- all state is self contained in the generated Future

no recursion: Future returned by an async function needs to have a fixed size known at compile time

nearly optimal: in terms of memory usage and allocations
- low overhead

when to use:
- need 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 if using a tiny fraction of the time slice
  - doing a lot of CPU work might prevent executor from running other tasks

similar tools in other languages

- Javascript: Promises and async/await
  - involves much more dynamic memory allocation, not as efficient
- Golang: goroutines are asynchronous tasks, but are not stackless
  - resizable stacks: possible because Go is garbage collected
  - runtime knows where all pointers are and can reallocate memory
- C++20: got stackless coroutines
  - still a lot of sharp edges

asynchronous H2O

for full implementation details, refer to tutorial 9

checking concurrent programs

increase confidence in code written:
- unit tests
- sanitizers
- model checking: mathematically proving that the code is correct

formal methods:

1. build model using Domain Specific Language (DSL)
2. check the model:
   - are all the constraints met?
   - does anything unexpected happen?
   - does it deadlock?

why invest time:
- check things make sense before starting (costly) implementation
- prove certain properties for existing code
- allow for aggressive optimizations without compromising correctness

advantages	**disadvantages**
• rigorous • verify all traces exhaustively • produce a system run that violates the requirements	• specification used can be faulty • tedious to come up with • time consuming

types of model checking:
- formal specification system
  - verify using model checker or proof assistant
  - use specification to write code (automatically or manually)
- add formal specification in code (invariants) as comments
  - use model checker to check invariants
  - difficult to make the model checker understand the code (some symbolic execution)
  - limited functionalities for concurrent code

model checkers:
- TLA$^+$ (TLC): Temporal Logic of Actions+
  - focuses on temporal properties
  - good for modelling concurrent systems (and distributed systems)
- Coq proof assistant
  - generates OCaml, Haskell, and Scheme
  - good for interactive proof methods
- Alloy (alloy analyzer)
  - focuses on relational logic
  - good for modelling structures

challenges:
- no global clock, no ordering of events
  - events happen at different times with different interleavings possible
  - participants see different interleavings
  - use a logical clock (happens-before) - memory model needed
- resilience - consistency - consensus
  - possible errors: deadlocks, livelock, starvation, lack of consensus
  - knowing classical synchronization problems might help

TLA$^+$

for full TLA$^+$ syntax, refer to lecture 12 slides

high-level language for modelling programs and systems, especially concurrent and distributed ones
- based on the idea that the best way to describe things precisely is with simple mathematics
- proposed by Leslie Lamport in 1999

approach:
1. write specification in TLA$^+$
2. specification is proven using a checker by exhaustively testing the states
3. manually write the code based on the TLA+ spec

behind the scenes:
- model checker finds all possible system behaviors (states) up to some number of execution steps
- examines the states for violations of desired invariance properties such as safety and liveness
- TLA$^+$ specifications use basic set theory to define safety and temporal logic to define liveness
  - safety: bad things won't happen
  - liveness: good things eventually happen