```
Concurrency is about dealing with lots of things at once.
Parallelism is about doing lots of things at once.
Not the same, but related.
One is about structure, one is about execution.
Concurrency provides a way to structure a solution to solve a problem that
may (but not necessarily) be parallelizable.
Rob Pike
```

Parallel Libs/Frameworks

- Cilk Plus -> install with custom gcc branch or custom llvm branch
  - https://gcc.gnu.org/git/gcc.git/
  - https://cilkplus.github.io/
- Threading Building Blocks - brew install tbb
- OpenMP - brew install gcc —-without-multilib
- CUDA - download installer from nvidia
- OpenCL - https://developer.apple.com/opencl/
  - included on mac OS after 10.7, integrates with GCD
  - https://developer.apple.com/library/content/documentation/Performance/Conceptual/OpenCL_MacProgGuide/Introduction/Introduction.html
- OpenGL - https://developer.apple.com/opengl/
- HPX - https://github.com/STEllAR-GROUP/hpx
- macOS Frameworks
  - Accelerate
  - Metal
  - OpenCL
  - OpenGL
  - NetFS
  - GLUT
  - GlKit

Additional Resources

- http://parlab.eecs.berkeley.edu/wiki/patterns/patterns
- https://bitbucket.org/VictorEijkhout/hpc-book-and-course
- https://bitbucket.org/VictorEijkhout/parallel-computing-book

Patterns for Parallel Programming

Finding Concurrency

Task Decomposition

Data Decomposition

Group Tasks

Order Tasks

Data Sharing

Design Evaluation

Algorithm Structure

Task Parallelism

Divide and Conquer

Geometric Decomposition

Recursive Data

Pipeline

Event-Based Coordination

Supporting Structures

SPMD

Master/Worker

Loop Parallelism

Fork/Join

Shared Data

Shared Queue

Distributed Array

Other

- SIMD
- MPMD
- Client-Server Computing
- Concurrent Programming with Declarative Languages

Implementation Mechanisms

UE Management

Thread Creation/Destruction

Process Creation/Destruction

Memory Synchronization and Fences

Barriers

Mutual Exclusion

Message Passing

Collective Communication

Other Communication Constructs

Structured Parallel Programming

Models

- Cilk Plus
- Threading Building Blocks
- OpenMP
- Array Building Blocks (ArBB)

Classification

- Flynn's Taxonomy
    - SISD

    - SIMD
    - MIMD
- shared memory
- distributed memory
- SIMT (Single Instruction, Multiple Threads)

Map

Collectives

Data Reorganization

Stencil and Recurrence

Fork-Join

Pipeline

Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects: 2

Reactor pattern

Proactor pattern

Asynchronous Completion Token

Acceptor-Connector

- http://www.cs.wustl.edu/~schmidt/POSA/POSA2/event-patterns.html

Python

Coroutines

Consider Coroutines to run many functions concurrently

- problems with threads
  - require special tools for safe coordination which makes them more difficult to reason about, difficult to extend, and harder to maintain
  - require a lot of memory (~8MB per thread)
  - costly to start
- coroutines are implemented as an extension to generators and cost a function call to start (functions have low overhead in python)
- allow code consuming generator to pass values back into the generator with each call using the `.send()` method
- initiate with a call to next
- `yield from` expression allows composition of generator coroutines, facilitating composition of larger coroutines from smaller ones
- Python 2 requires extra work and has additional limitations
- powerful tool for separating core logic of program from interaction with surrounding environment

Generators

- coroutines via advanced generators (PEP 342) added in 2.5 which augments generators to have .send() method
- augmentation is how coroutines were born
  - generators produce data for iteration
  - coroutines are consumers of data
  - don't mix the two concepts (avoids complex reasoning)
  - coroutines are related to iteration
  - use of using yield to produce a value in a coroutine but not tied to iteration

concurrent.futures

- use for true parallelism
- multiprocessing accessed via concurrent.futures
- runs additional interpreters as child processes to utilize multiple CPU cores
- each has separate GIL and can fully utilize one CPU core
- maintains link to parent, receives instructions and returns results

- ThreadPoolExecutor still bound by GIL

```
pool = ThreadPoolExecutor(max_workers=2)
results = list(pool.map(work, values))
```

- ProcessPoolExecutor will speed up considerably

```
pool = ProcessPoolExecutor(max_workers=2)
results = list(pool.map(work, values))
```

- underlying actions from multiprocessing of ProcessPoolExecutor
  i. reads each item from the input collection passed to map
  ii. serializes data using pickle
  iii. copies data from parent to child via a local socket
  iv. child deserializes
  v. child imports work function and runs (parallel to other child processes)
  vi. serializes result to bytes, transfers back to parent
  vii. deserializes in parent and merges results into a single list
- well suited to small, isolated functions with no state sharing and small amounts of data passed from parent to children
- access through concurrent.futures and leave remaining multiprocessing components unless prepared to deal with complexity

socket

select

epoll

threading

Use threads for blocking I/O and not parallelism

- CPython compiles source to byte(word)code and then interprets in a stack-based interpreter
- interpreter maintains state during a run, ensures coherence using GIL
- GIL is a mutex lock that prevents preemptive multithreading
  - thread taking control by interrupting another thread
- result of GIL is that multiple threads still means only one progresses at a time
- multiple threads can actually take more time in Python
- still useful to allow multiple functions to make progress without manually juggling because GIL ensures a certain amount of fairness between threads

- threads also help with blocking I/O by protecting program from hanging while waiting for system calls that result in blocking I/O

```
thread = Thread(target=slow_systemcall)
thread.start()
```

- Python threads release the GIL just before they make system calls and reacquire them as soon as they're done which means the system calls can run in parallel

Use Lock to prevent data races in threads

- still need to lock data access between threads despite GIL because interpreter instructions can be interrupted mid-data access
- use the tools in the threading built-in module
    - Lock is the standard mutual-exclusion lock

```
lock = Lock()
with lock:
    # access data
```

Use Queue to coordinate work between threads

- useful arrangement of concurrent work is pipeline of functions
- good for work that includes blocking I/O or subprocesses
    - easily parallelizable activities in Python
- with producer/consumer pipeline and custom queue/lock combo, varying speeds of worker functions can cause back up
- Queue class in queue built-in blocks on calls to get until new data is available
- Queue class allows specification of maximum amount of pending work allowed between two phases
- can track progress of work using task_done, eliminates need for polling
- issues with concurrent pipelines
    - busy waiting
    - stopping workers
    - memory explosion
- Queue has required facilities for building concurrent pipelines
    - blocking operations
    - buffer sizes
    - joining

asyncio

- added in 3.4, built on coroutines
    - must use `yield from`

    - must invoke with `yield from` or pass to asyncio function
    - should use @asyncio.coroutine decorator
- Trollius backports to 2.6 with specific changes
- don't use time.sleep() in aysncio coroutines unless want to block the main thread

```
yield from asyncio.sleep(DELAY)
```

- differences between threads and aysncio
    - asyncio.Task ~= threading.Thread
    - Task is like a green thread in libraries that implement cooperative multitasking (gevent)
    - Task runs a coroutine and a Thread invokes a callable
    - don't manually create Tasks
        - coroutine -> asyncio.async()
        - coroutine -> loop.create_task()
    - Tasks are already scheduled to run
    - Threads must be `start` ed
    - threads work with standard functions, asyncio works with coroutines using `yield from`
    - threads do not have an API to cancel, tasks have Task.cancel() which raises CancelledError inside coroutine
    - outer coroutine must be executed with loop.run_until_complete in the main function
- coroutines in asyncio are protected against interruption by default
- must explicitly yield to allow rest of program to run (synchronized by definition)

asyncio.Future

- similar interface to concurrent.futures.Future but not interchangeable
- BaseEventLoop.create_task() takes coroutine and schedules to run
- Task is subclass of Future, designed to wrap a coroutine
- analogous to concurrent Futures (created through Executor.submit())
- methods
    - done
    - add_done_callback
    - result
- designed to work with yield from
    - don't need .add_done_callback() -> put processing after yield from
    - don't need .result() -> result of a yield from expression is the result

```
result = yield from future
```

- TODO: more here

subprocess

Use subprocess to Manage Child Processes

- good for gluing command line utilities together
- Python is CPU bound but spinning up subprocesses allows for CPU-intensive workloads
- subprocess replaces
    - popen
    - popen2
    - os.exec*

```
proc = subprocess.Popen(['cmd'], stdout=subprocess.PIPE)
out, err = proc.communicate()
# out.decode('utf-8')

proc = subprocess.Popen(['cmd'])
while proc.poll() is None:
    # do something
proc.poll() # 0
```

- run in parallel with Python interpreter
- can write to child process stdin
- use timeout parameter with communicate to avoid deadlocks and hanging child processes

C extensions

- can be used to implement performance-critical code without thread limitations of GIL (also runs faster)

- adds cost of time and comprehensibility

Unix

POSIX Threads

Thread Synchronization

Critical Sections and Semaphores

POSIX IPC

Asynchronous Events

Signals

Timers

Communication

Connection-Oriented Communication

- The Client-Server Model
- Communication Channels
- Connection-Oriented Server Strategies
- Universal Internet Communication Interface (UICI)
- UICI Implementations of Different Server Strategies Section
- UICI Clients
- Socket Implementation of UICI
- Host Names and IP Addresses
- Thread-Safe UICI

Connectionless Communication and Multicast

- Simple-Request Protocols
- Request-Reply Protocols
- Request-Reply-Acknowledge Protocols
- UDP vs TCP
- Multicast

Alternative I/O Models (basis for other mechanisms)

- Level-triggered
- Edge-Triggered

I/O Multiplexing

- select
- poll

Signal-Driven I/O

- SIGIO and signal handlers

epoll (Linux)

- kqueue

Waiting on Signals and File Descriptors

- pselect
- the self-pipe trick

C++

Threads

Synchronization

Futures

Promises

Atomics

Custom thread pooling

Message passing frameworks

- ZeroMQ

The Art of Multiprocessor Programming

Mutual Exclusion

Concurrent Objects

Foundations of Shared Memory

Primitive Synchronization Operations

Consensus

Spin Locks and Contention

Monitors and Blocking Synchronization

Linked Lists

Concurrent Queues and the ABA Problem

Concurrent Stacks and Elimination

Counting, Sorting and Distributed Coordination

Concurrent Hashing and Natural Parallelism

Skiplists and Balanced Search

Priority Queues

Futures, Scheduling, and Work Distribution

Barriers

Transactional Memory

macOS

Unix Sockets and Networking

CFRunLoop, CFSocket and Events

kqueue and FSEvents

Bonjour (Zeroconf standard)

Multiprocessing

- scheduling
- fork
- exec
- Pipes

NSTask

NSProcessInfo

NSTask

NSFileHandle

NSPipe

Multithreading

Posix Threads

Cocoa and Threading

- NSThread

Operations

NSOperationQueue

Grand Central Dispatch

Queues

Dispatching

Memory management

SCMSW

TODO: go back and do the self-study sections and add notes here

- concurrency - dealing with lots of things at once
- parallelism - doing lots of things at once

Threads and Locks

Mutual Exclusion and Memory Models

- creating a thread

```
Thread t = new Thread() {
    public void run() {
        // do something
    }
};

t.start();
// Thread.yield();
// t.join():
```

- yield is a hint to the system scheduler that the thread is willing to yield its use of a system processor
- not calling yield after creating a thread adds startup overhead
- read-modify-write pattern is common
- intrinsic lock ~= mutex ~= monitor ~= critical section

- intrinsic locking

```
// synchronized method
public synchronized void increment() { ++count; }

// synchronized block
synchronized (var) {
    // do something with var
}
```

- race conditions are primary concern -> results in non-deterministic behavior
- remember
  - compiler optimizations can reorder instructions
  - JVM optimizations can reorder instructions
  - hardware optimizations can reorder instructions
- memory visibility - the visibility of memory changes between threads is defined by the Java memory model
  - no guarantees unless using synchronization in BOTH threads
- to avoid the possibility of deadlock, always lock multiple locks in a globally defined and consistent order
- do not use Java's hashCode method for ordering locks because it is not guaranteed to be unique
- deadlock - occurs when a process or thread enters a waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process
- summary:
  - synchronize all access to shared variables
  - use synchronization in both the writing and reading threads
  - acquire multiple locks in a fixed global order
  - don't call alien (unknown code) methods while holding a lock
  - hold locks for the shortest possible amount of time
- memory model
- double checked locking anti-pattern
- limits of intrinsic locks
  - no way to interrupt a blocked thread
  - no way to time out while trying to acquire an intrinsic lock
  - can only acquire with a synchronized block/method (lock acquisition and release must occur in the same method and must be strictly nested)
- ReentrantLock is the solution
  - explicit locking
  - interruptible locking
  - timeouts

- the only way for a thread to exit in Java is for the run method to complete, so a deadlocked thread cannot be interrupted

```
final l = new ReentrantLock();

Thread t = new Thread() {
    public void run() {
        try {
            l.lockInterruptibly();
        } catch (InterruptedException e) {}
    }
}
```

- timeouts

```
if (l.tryLock(1000, TimeUnit.MILLISECONDS)) {
    // lock acquired
} else {
    // unable to lock
}
```

- livelock - similar to deadlock, except that the states of the processes involved constantly change with regard to one another with neither progressing
- livelock is a risk with some algorithms that detect and recover from deadlock
    - if more than one process takes action, the deadlock detection algorithm can be repeatedly triggered
    - can be avoided by ensuring that only one process (chosen arbitrarily or by priority) takes action
- deadlock - a situation in which two or more processes are unable to proceed because each is waiting for one of the others to perform some action
- livelock - a situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work
- starvation - a situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen
- hand-over-hand locking - used for sequenced containers (esp. linked list)
    - don't lock full container
    - iterate through container, locking nodes/elements on either side of the target element before locking and accessing the target element, release locks and move forward
- condition variables - synchronization primitive that is associated with a lock and used to signal to other threads the state of that desired condition
- a condition variable is associated with a lock and a thread must hold that lock before being able to wait on that condition
    - after the lock is acquired, the condition is checked and the thread either progresses or waits on the condition

- atomic operation - any operation that, from the point of view of another thread, appears to be a single operation (though it may be many collected into one) e.g. it can never be interrupted when partially complete

```
//
condition.signal()
//
l.lock();
try {
    while (condition) {
        condition.await();
    }
} finally {
    l.unlock();
}
```

- Java has atomics which provide sets of atomic operations on various types (generally primitives) that do not require locks
- atomics are the foundation of lock-free and non-blocking algorithms
- the volatile keyword in Java (unlike C++) ensures that reads and writes to that variable will not be reordered
- summary:
    - use ReentrantLock and java.util.concurrent.atomic to allow a thread to
        - be interrupted while trying to acquire a lock
        - time out while attempting to acquire a lock
        - acquire and release locks in any order
        - use condition variables to wait for arbitrary conditions
        - avoid locks entirely by using atomic variables
- use Executors.newFixedThreadPool(size) to create an ExecutorService thread pool and automatically queue excess thread creation requests, thus reducing risks of issues like DOS
- create thread pools the size of the number of CPUs on a machine or 2x for hyperthreaded
- copy-on-write - paradigm that makes a copy of a data structure whenever it is changed
    - don't forget this will invalidate iterators
    - only appropriate for certain use cases
    - more efficient than making a defensive copy each time
- for certain use cases a blocking queue is better than a non-blocking unbounded Queue
- summary:
    - use thread pools instead of creating threads directly
    - for listener management, consider copy-on-write data structures

    - use blocking queues for the producer-consumer pattern to allow producers to get ahead of consumers but limit this amount
    - use concurrent hash maps for concurrent access to maps
- questions:
    - how does a ForkJoinPool differ from a thread pool?
    - what is work stealing and when is it useful?
    - difference between CountDownLatch and CyclicBarrier?
    - Amdahl's Law?
- overview:
    - strengths
        - broad applicability (coarse to fine grain granularity of solutions for problems)
        - close to the metal and thus generally efficient
        - easily integrated into many different languages with different paradigms
    - weaknesses
        - no direct support for parallelism (can be used to parallelize a sequential algorithm, but solution has to be constructed from concurrent primitives, which can result in non-determinism)
        - generally support only shared memory architectures (need an alternate solution for distributed memory)
        - difficult to program with threads and locks
        - the difficulty generally lies in reasoning about what is happening and, more seriously, testing

Functional Programming

- Clojure - dialect of Lisp that runs on the JVM
- functional programming is partially based on an (sometimes complete) avoidance of (shared) mutable state (be aware of pure vs. impure functional languages and the relationship to the distinction of functions w/wout side effects)
- removing shared mutable state makes concurrency much easier (to write, test and reason about)
- use fold from clojure.core.reducers for a parallel version of reduce
- lazy sequences - elements are generated only when they are needed
- summary:
    - avoid shared mutable state to simplify concurrency
    - map & mapcat - apply a function to every element of a sequence

    - laziness allows large or infinite sequence manipulation, only evaluating elements when needed
    - use reduce (or parallel fold) to reduce a sequence to a single value
- pmap is like map but parallel and semi-lazy, i.e. parallel computation stays ahead of consumption but it will not the entire result unless it is required
- reducer - recipe for how to reduce a collection
    - map - function and (possibly lazy) sequence -> (possibly lazy) sequence
    - reducer returns the recipe for creating the result
        - more efficient than a chain of functions returning lazy sequences because no intermediate need to be created
        - allows fold to parallelize the entire chain of operations on the underlying collection
- Clojure has protocols which are similar to Java's interfaces
- binary chop
- fold tree
- summary:
    - use pmap to parallelize map and return a semi-lazy parallel map
    - parallel map can be batched for efficiency with partition-all
    - fold parallelizes reduce oprations with an eager divide-and-conquer strategy
    - clojure.core.reducers versions of map, mapcat, and filter returns reducibles which can be thought of as recipes for how to reduce a sequence
- functional programming focuses on expressions and thus allows manipulation of the order of evaluation
    - much more declarative, statement of what result should be and method for evaluation is left to engine
- referential transparency - anywhere an invocation of the functions appears it can be replaced with result without changing the behavior of the program
- future - body of code that executes in another thread and returns a future object
- promise - value that is realized in a separate thread and blocks until it is realized
    - does not cause code to be run, but value is set with deliver
- summary:
    - functions in a functional language are referentially transparent
    - expression evaluation order can be safely modified
    - facilitates dataflow style of functional programming -> code executes when data it depends on becomes unavailable

- overview:
    - some concurrent programs always imply non-determinism, but not all parallel programs imply non-determinism
    - most race conditions that occur in traditional thread and lock-based programming occur because of the particular implementation of the solution rather than an intrinsic non-determinism in the problem
    - referential transparency allows for modification of order of evaluation and thus parallelization in an almost trivially easy way
    - monads and monoids are used in statically typed functional languages like Haskell to encode where particular functions and expressions can be evaluated to keep track of side effects while remaining functional
    - strengths:
        - confidence, simpler to reason about
        - referential transparency and elimination of mutable state
    - weaknesses:
        - efficiency (but generally less than predominantly reported, and worth it for gain in robustness and scalability)

The Clojure Way - Separating Identity from State (Functional Concurrency, Persistent Data Structures, Software Transactional Memory)

- certain concurrent solutions require shared mutable state
- Clojure is impure - provides a number of types with concurrency-aware mutable variables
- impure functional vs. imperative
    - imperative - mutability is default and code modifies them frequently
    - impure functional - immutable by default and code only modifies mutable variables when absolutely necessary
- atoms - similar to atomic variables (built on Java's java.util.concurrent.atomic in Clojure)
    - swap to set
    - can be any type
- persistent data structures - behave as though each modification results in a complete copy (not related to persistence on disk)
    - implemented behind the scenes with structure sharing

- !!! Structure sharing
- all Clojure collections are persistent
- persistent data structures can be implemented in non-functional languages but require work to ensure the contract
- for concurrency, pds's essentially result in a copy (similar to thread local) for each thread
- pds's separate identity from state
- variables in imperative languages complect (interweave, interconnect) identity and state
- retrieving the current state associated with an identity, for pds's, results in an instance of an immutable data structure
- summary:
    - diff between impure functional and imperative
    - persistent data structures in functional languages allow the separation of identity and state
- agents are like atoms and are modified with send
    - takes function and optional arguments but unlike atoms send returns immediately and functions passed are serialized when called by different threads so side effects can occur
- agents vs. actors
    - agent has a value that can be retrieved directly with deref
    - actors encapsulate state but provide no direct means to access
    - actors encapsulate behavior
    - agents are passed the function that performs the behavior
    - actors provide sophisticated support for error detection and recovery
    - actors can be remote
    - composed actors can deadlock
    - composed agents cannot
- in-memory logging can be very helpful for debugging and understanding threaded programming
    - difficult with threads and locks
    - trivial with agents
- refs provide software transactional memory
    - allows coordinated changes to multiple values at once (much like database transaction and record modifications)

- STM transactions are atomic, consistent, and isolated
    - atomic - all side effects (results) of transaction occur or none of them do
    - consistent - validators for transaction must all pass or transaction will not be committed
    - isolated - multiple transactions can execute concurrently but the effects of those transactions are indiscernible from running those transactions sequentially
- Clojure allows the isolation constraint to be relaxed
- summary:
    - atoms enable independent, synchronous changes to single values
    - agents enable independent, asynchronous changes to single values
    - refs enable coordinated synchronous changes to multiple values
- skipped this section
- summary:
    - generally simple atom-based concurrency is enough
    - STM-based code in which multiple refs are coordinated through transactions can be transformed into an agent based solution with the refs consolidated into a single compound data structure accessed via agents
    - make choice based on performance characteristics
- overview:
    - strengths
        - builds on general functional strengths
        - Clojure persistent data structures allow mutable variables to separate identity and state
    - weaknesses
        - no support for distributed programming
        - no support for fault tolerance

Actors

- retains mutable state but does not share it
- objects that encapsulate state and communicate via actual messages but run concurrently with each other
- used in many languages but associated with Erlang
- Elixir (and Erlang) are dynamically typed impure functional languages (Elixir runs on the Erlang virtual machine)
- actor is called a process in Elixir and Erlang
    - process in this context is more lightweight than system threads

- define actor, spawn instance, store process identifier to use for sending messages
- messages are sent asynchronously by being placed in a mailbox (which is just a queue)
- actors sit in loops waiting for messages to arrive and match the message
  - the loop recursion does not blow up stack because Elixir uses tail-call elimination
- actors have both a way to trigger stop and to register for notifications on stop
- can use recursion to maintain state in actor messaging
- common practice to wrap messaging and handling in APIs
- can build support for replies
- can name processes rather than work with process identifiers
- summary:
  - create a new process with spawn()
  - send a message to a process with send()
  - use pattern matching to handle messages
  - create a link between two processes and receive a notification when one terminates
  - implement bidirectional, synchronous messaging on top of the standard asynchronous messaging
  - register a name for a process
- define actors with state to cache
- use links to propagate abnormal termination (remember that they are bidirectional)
  - normal termination does not propagate the termination
- system process - can trap another process' exit by setting :trap_exit flag
- can supervise processes that monitors one or more worker processes and take appropriate action on fail
- can use after clause to implement timeouts
- message delivery guarantees
  - when nothing breaks
  - will be notified on break as long as registered via links or monitors
- the error-kernel pattern
  - TODO:
- actor patterns avoid defensive programming and prescribe to "let it crash" philosophy
  - simpler, clear separation between happy path and fault-tolerant code
  - separate and no shared state
  - supervisor can fix and log errors
- summary:
  - links are bidirectional
  - links propagate errors
  - system processes exit with exit message
- actor model supports distribution
- OTP library (Open Telecom Platform but just OTP) is a powerful library for Erlang and Elixir
  - better restart logic
  - debugging and logging

    - hot code swapping
    - more
- provides a supervisor with different restart strategies
    - one-for-one
    - one-for-all
- instance of Erlang VM is a node and can create and connect multiple nodes via IP
    - remote execution and messaging
    - need to pay careful attention to security and many other cluster design trade-offs
- see section for fault tolerance info
- summary:
    - clusters of nodes
    - actors can interact between nodes
    - allows distributed systems and failure recovery
- overview
    - strengths
        - messaging and encapsulation
        - fault tolerance
        - distributed programming
    - weaknesses
        - susceptible to deadlock
        - susceptible to failure modes specific to actors
            - mailbox overflow
        - actors provide no direct support for parallelism
            - needs to be built from building blocks
        - not suitable for fine grained parallelism
- see Akka for JVM-based actor solutions

Communicating Sequential Processes

- similar to actors but focus is on channels over which messages are sent rather than the entities and the messages
- channels are first class and replace mailboxes
    - can be independently
        - created
        - written to
        - read from
        - passed between processes
- old idea experiencing a resurgence due to Go
- core.async in Clojure
- channel = thread safe queue
- senders don't have to know about receivers

- by default channels are synchronous or unbuffered
  - writing to a channel blocks until it is read from
- can create a buffered channel by passing a buffer size to chan
- channels can be closed
- can write collections into channels with utility functions
- dropping buffers
- sliding buffers
- Clojure thread macro uses a CachedThreadPool under the hood to avoid the startup costs of creating threads
- can resolve issues of blocking buffers with event system but this leads to excess of global state and makes code harder to reason about due to fragmented control flow
- Go blocks are the alternative that allow event driven code without sacrificing structure or readability
- Lisp-like languages have macros
- go macro transforms block into state machine
  - rather than blocking, thread parks, relinquishing control of current thread
  - when next able to run, performs a state transition and continues execution, potentially on another thread
    - reading the value from the channel
  - allows runtime to multiplex many go blocks over a limited thread pool
- blocking in a go block just blocks the current thread for the go block
- go blocks are for cheap efficiency
- summary:
  - channels are synchronous (unbuffered) by default
  - various buffering strategy (full buffer, sliding buffer, dropping buffer)
  - go blocks are macros that invert control flow to generate a state machine for thread management via a thread pool to execute the related block of code
  - blocking and parking channel operations
- alt! allows for management of multiple channels at once
- timeout function returns a channel that closes after a certain number of milliseconds
- using reified timeouts simplifies management of connection semantics
- can use the timeout to implement polling (better as a macro)
- asynchronous i/o can be challenging
  - in Clojure, need to replace callbacks with put to fire and forget writes to channels
- summary:
  - channels and go blocks allow for efficient asynchronous code that reads naturally without complexity of callbacks
  - existing callback-based APIs can be utilized with a minimal callback that writes to a channel
  - alt! allows a task to read from, write to multiple channels
  - the timeout function returns a channel that closes after an interval - allowing timeouts to be treated as first class entities

    - parking calls should be contained within a go block
    - Clojure macros can be used to inline code
- ClojureScript cross compiles to JavaScript and supports core.async
- browser UI works on single thread but ClojureScript's appearance of multithreading results in a form of cooperative multitasking
    - improves code structure and clarity
- WebWorkers can be accessed in ClojureScript via the Servant library
- provides means for event handling and taming callbacks
- summary:
    - ClojureScript cross compiles to JavaScript
    - core.async in client-side development
    - cooperative multitasking in single-threaded JavaScript environments
    - relief from callback-based programming issues
- overview:
    - strengths:
        - more flexble than actors (medium of communication is not tightly coupled)
        - go macro inversion of control brings a dramatically improved programming model for traditionally callback-based (multithreaded) application areas (UI, async IO, etc)
    - weaknesses:
        - actor model is based on awareness of distribution and fault-tolerance (not CSP)
        - susceptible to deadlock with no direct support for parallelism

Data Parallelism (GPGPU Programming)

- GPGPU - general purpose computing on the GPU
- many technologies
    - using OpenCL (Open Computing Language)
- graphics cards are tuned to manipulate large amounts of data quickly (tuned for data parallelization)
- many methods for data parallelization
    - pipelining
        - simple operations like addition are actually several machine instructions
        - pipelining takes advantage of improvements in number of operations for multiple data
    - multiple ALUs (Arithmetic Logic Unit)
        - coupling multiple ALUs to a wide memory bus allows simultaneous operations on multiple data
    - these two are combined with many other techniques to achieve the end speed-up in GPUs
- OpenCL is C-like and targets multiple architectures by hooking into compiler
- OpenCL operates on work-items and task as a programmer is to divide code into smallest work-items possible

- optimizing OpenCL can be difficult and require architecture-specific considerations
- specify OpenCL processing by writing kernels (essentially OpenCL functions)
    - create a context and a command queue
    - compile the kernel
    - create buffers for input and output data
    - enqueue a command that executes the kernel once for each work-item
    - retrieve the results
- OpenCL has a profiling API
- multiple return values
- error handling
- summary:
    - OpenCL parallelizes a task by dividing it into work-items
    - specify processing for each work-item by writing a kernel
    - follow process above for executing a kernel
- provides a number of APIs for querying device info, platform parameters, and many other API objects
- can target more than just GPUs (CPUs, OpenCL accelerators)
- GPUs can have more than one compute unit
- GPUs have specified memory
- platform model
    - host connected to one or more devices
    - device has one or more compute units
    - compute units provide a number of processing elements
    - work-items execute within processing elements
    - a collection of work-items within a compute unit is a work-group
- memory model:
    - global memory - available to all work items on a device
    - constant memory - region of global memory that remains constant during kernel execution
    - local memory - memory local to a work-group
    - private memory - private to a single work-item
- data parallel reduce
- barriers
    - a synchronization primitive that when executed by a single unit of execution, requires that all other grouped units of execution must complete the same section of code (block, barrier, work, etc) before any can proceed beyond that point
    - commonly known as a rendezvous
- in reduce, barrier ensures
    - no work-item begins reducing before all have copied memory from global to local

    - OpenCL only ensures relaxed memory consistency (local memory changes are not guaranteed to be visible to other units of execution), so barriers guarantee the local memory changes will be visible to others at points like the end of a loop
- need to create local buffer to execute kernel
- summary:
    - work-items execute on processing elements
    - processing elements are grouped into compute units
    - a group of work-items executing on a single compute unit is a work-group
    - work-items in a work-group communicate through local memory using barriers to synchronize and ensure consistency
- Java with Lightweight Java Graphics Library (LWJGL) provide wrappers for OpenCL and OpenGL
- OpenCL kernel on GPU can operate directly on OpenGL buffers
- OpenGL 3D scene is constructed from a mesh of triangles
    - vertex buffer defines a set of vertices (points in 3D space) (`[0, 0, 1, 0, 0, 2, 0, 0, 0, 1, 0, 1, ...]`)
    - index buffer defines which of those vertices is used to draw each triangle (`[0, 3, 1, 4, 2, 5, ...]`)
    - each buffer has an IDa and is bound to a target
- to access an OpenGL buffer from an OpenCL kernel
    - acquire with clEnqueueAcquireFLObjects()
    - set it as an argument to kernel
    - release buffer
    - wait for processing to finish
- see book for example to simulate ripples
- summary:
    - OpenCL kernel on GPU can operate directly on OpenGL buffers
    - create an OpenCL view of an OpenGL buffer with clCreateFromGLBuffer()
    - acquire an OpenGL buffer before passing it to a kernel with clEnqueueAcquireGLObjects()
    - release the buffer after the kernel has finished with clEnqueueReleaseGLObjects()
- overview:
    - strengths:
        - good for processing large amounts of numerical data (fluid dynamics, finite element analysis, n-body simulation, simulated annealing, ant-colony optimization, neural networks, etc)
        - GPUs have many benefits
            - powerful data-parallel processors
            - power consumption (better GFLOPS/watts than traditional CPUs)
    - weaknesses:
        - use for nonnumerical problems (e.g. natural language processing) is not straightforward

        - optimization of OpenCL kernels depends on a knowledge of the underlying architecture
    - other GPGPU options
        - CUDA
        - DirectCompute
        - RenderScript Computation

Lambda Architecture (and MapReduce)

- combines large-scale batch processing of MapReduce with real-time responsiveness of stream processing
    - scalable
    - responsive
    - fault-tolerant
- Nathan Marz, many facets, focus on parallel and distributed aspects
    - batch layer
    - speed layer

```
raw data -> batch layer -> batch views    -> results
         -> speed layer -> realtime views ->
```

- builds upon MapReduce
- MapReduce
    - name for pattern
        - map - map over a data structure
        - reduce - perform a reduce operation
    - system for efficient distribution of map and reduce operations on large data sets across a cluster of computers
- name from deep similarities between functional programming and the architecture
- popularized at Google and then with Hadoop
- maintaining Hadoop clusters is notoriously time consuming
- many providers offer managed Hadoop clusters
    - Amazon EMR - elastic MapReduce
    - Horton Works
    - Cloudera
    - MapR
- input is split into files and fed first to Mappers, joined through a single Mapper, then Reducers, then joined through Reducers (all distributed across machines)
- requires a Hadoop tool to run and can be run locally
- Hadoop guarantees that keys will be sorted before being passed to Reducers
- EMR reads from and to S3 by default

- additional benefits beyond speed with distributed parallelization
  - failure is a guarantee with large clusters and Hadoop is built with fault-tolerance in mind
  - handles failure retries and data management to avoid loss of data using HDFS and replication
  - manages transfer of data between memory and HDFS during execution
- summary:
  - splitting problems into map stage and reduce stage simplifies parallelization
  - Hadoop
    - splits input into distributed Mappers which output key/value pairs
    - sends to Reducers which output final output generally as key/value pairs
    - uses intelligent partitioning
- current volume of data being processed pushes traditional databases beyond point at which they were designed to function
- replication, sharding and other techniques allow DBs to scale but become more difficult as it scales
  - adds maintenance overhead
  - complexity
  - possibility of human error
  - reporting and analysis difficulty (warehousing and ETL issues)
- can divide data into raw data and derived information, and data is better raw (but clean) as it is built on immutable events
- batch views are precomputed snapshots of derived data and is the job of the batch layer
- batch views are run to completion each time (but can hand-roll an incremental algorithm, though benefits are questionable)
- usually a serving layer sits in front of batch view results to allow queries
  - some specialized DBs exist that optimize for random reads and only batch update writes (Voldemort, ElephantDB)
- summary:
  - raw data is immutable
  - derived data
  - resulting systems are
    - highly parallel and can handle TBs of data and more
    - simple and less error prone
    - tolerant to technical and human error
    - handle day-to-day ops and historical reporting and analysis
  - drawback is latency (accounted for by Lambda Architecture with speed layer)
- speed layer generates real-time views that, when combined with batch views, allow for up-to-date queries
- uses Storm to create the speed layer
- speed layer is more difficult to build than batch layer
  - however only needs to handle unprocessed portion of data

- can think of traditional DB as degenerate form of Lambda Architecture without running batch layer
    - synchronous (blocking) reads and writes
- better to use async approach with message queues
    - Kafka
    - Kestrel
    - Google Cloud Pub/Sub
    - Amazon Kinesis
- adds decoupling of clients
    - fewer clients can handle higher volume
    - spikes in demand would lead to timeouts etc in synchronous systems whereas async system falls behind and catches up when possible
    - stream processor can exploit parallelism, taking advantage of multiple compute elements to provide fault tolerance and improved performance
- need to consider expiring data
- Storm attempts to simplify real-time processing in the same way as MapReduce simplifies batch processing
- Storm processes named tuples
    - created in spouts
    - processed in bolts
    - spouts and bolts are connected in streams to create a topology
- run internally parallel using workers which are distributed and tuples and can be sent and processed to any in parallel
    - used for fault tolerance
- summary:
    - Storm is one means for implementing the speed layer in the Lambda Architecture
        - processes streams of tuples in real time
        - spouts and bolts in a topology
        - spouts and bolts have multiple workers that run in parallel and are distributed across the nodes of a cluster
        - Storm provides "at least once" semantics
- overview:
    - Lambda Architecture
        - raw data immutability (source of truth forever) is similar to separation of identity and state
        - MapReduce is like parallel functional programming
        - distributes processing over a cluster (like Actors) for performance and fault tolerance
        - streams of tuples is similar to message passing in CSP
    - strengths:
        - large quantities of data
        - reporting

        - analytics
        - (substitute for data warehousing)
    - weaknesses:
        - overhead of system is not worth it unless working with large amounts of data
    - alternatives
        - many batch layers aside from MapReduce
            - Spark - both batch and speed (streaming) layer

Future

- immutability
    - persistent data structures
    - raw data
    - pure and impure functional languages
    - messages (Actors and CSP)
    - threads and locks (simplifies code and logic)
- distributed
    - cores
    - clusters
- additional considerations
    - fork/join and work-stealing
        - Cilk
    - dataflow
        - poor general-purpose dataflow languages but still important
        - VHDL and Verilog
    - reactive programming
        - closely related to dataflow
        - programs automatically react to propagated changes
        - Microsoft Rx (Reactive Extensions) library + more
        - parallels Storm, Actors, and CSP
    - functional reactive programming
        - Elm + more
    - grid computing
        - loosely coupled distributed cluster
        - heterogeneous with fluid modification to cluster
        - SETI@Home
    - tuple spaces
        - can be used for IPC
        - first introduced to Linda coordination language

studentloans.gov

Actors

- CAF
- Akka

Futures

- Folly

Java

Green threads vs. lightweight processes vs. coroutines vs fibers vs protothreads

libuv

- cross-platform asynchronous I/O
- event loop backed by epoll, kqueue, IOCP, event ports.
- asynchronous TCP and UDP sockets
- asynchronous DNS resolution
- asynchronous file and file system operations
- file system events
- ANSI escape code controlled TTY
- IPC with socket sharing, using Unix domain sockets or named pipes (Windows)
- child processes
- thread pool
- signal handling
- high resolution clock

- threading and synchronization primitives

Clean Code

13. Concurrency

Why Concurrency?

- decoupling strategy
  - separate what gets done from when it gets done
- this decoupling can improve throughput and structure
- structure
  - units of execution resemble microcosmic instances of computers
  - not all tied to single main loop
- many cases are required due to response time and throughput constraints of systems

Myths and Misconceptions

- difficult to do correctly
- common myths
  - concurrency always improves performance
  - design does not change when writing concurrent programs
  - understanding concurrency issues is not important when working with a container such as a web container
- improvements
  - concurrency incurs some overhead (performance and development time)
  - correct concurrency is complex
  - concurrency bugs are not usually repeatable
  - concurrency often requires a fundamental change in design strategy

Challenges

- need to understand
  - compiler/JIT compiler
  - compiled output (e.g. byte code, etc)
  - language's memory model
- many possible paths of execution through even the simplest threaded systems
  - many may generate valid results

    - problem is that some don't

Concurrency Defense Principles

Single Responsibility Principle

- given class or method should have a single reason to change
- separate concurrency design from the rest of the code
    - strictly remove reliance on implementation details
- considerations
    - concurrency-related code has its own life cycle of development, change, and timing
    - concurrency-related code has its own challenges which are different from and more difficult than those of nonconcurrent code
    - the number of ways in which miswritten concurrency-based code can fail makes it challenging enough without the added burden of surrounding application code
- recommendation
    - keep your concurrency-related code separate from other code

Corollary: Limit the Scope of Data

- use mutual exclusion (e.g. synchronized and ReentrantLock in Java) to protect critical sections that access any shared objects
- it is important to restrict the number of shared objects and related critical sections
- more shared data increases the likelihood that
    - users of that data will forget to protect one or more accesses to that data, breaking other code that modifies that shared data
    - there will be duplication of effort required to make sure everything is effectively guarded (violation of DRY)
    - it will be difficult to determine the source of failures, which are already hard enough to find
- recommendation
    - take data encapsulation to heart and severely limit the access of any data that may be shared

Corollary: Use Copies of Data

- avoid sharing data by copying objects and either
    - treat them as read-only

    - modify copies in individual threads and merge results after use
- best to benchmark to determine cost of copying objects
    - avoidance of synchronization overhead might make up for costs

Corollary: Threads Should Be as Independent as Possible

- write threaded code such that each thread exists in its own world, sharing no data with any other thread
    - each thread processes one client request, with all of its required data coming from an unshared source and stored as local variables
- makes each thread behave as if it were the only thread in the world and there are no synchronization requirements
- recommendation
    - attempt to partition data into independent subsets than can be operated on by independent threads, possibly in different processors

Know Your Library

- use the provided thread-safe collections
- use the executor framework for executing unrelated tasks
- use non-blocking solutions when possible.
- several library classes are not thread safe
- in Java
    - java.util.concurrent
        - ConcurrentHashMap
        - ReentrantLock
            - can be acquired in one method and released elsewhere
        - Semaphore
            - a lock with a count
        - CountDownLatch
            - waits for events before releasing
            - allows all threads to have a fair chance of starting at about the same time
- recommendation
    - review the classes available to you
    - in Java
        - java.util.concurrent
        - java.util.concurrent.atomic
        - java.util.concurrent.locks

Know Your Execution Models

- different methods for partitioning behavior in concurrent applications
- most concurrent problems encountered will likely be some variation of these three problems

- recommendation
  - learn these basic algorithms and understand their solutions

Definitions

- bound resources
  - resources of a fixed size or number used in a concurrent environment
  - e.g. database connections, fixed-size read/write buffers
- mutual exclusion
  - method to ensure only one thread can access shared data or a shared resource at a time
- starvation
  - situation in which one thread or a group of threads is prohibited from proceeding for possibly forever
  - e.g. always allowing fast threads to run first may keep slower threads from running
- deadlock
  - two or more thread waiting on each other where each has a resource that the other requires and neither can progress until it gains access to that resource
- livelock
  - threads actively competing with each other to make progress but being delayed (possibly forever) due to the other thread impeding that progress

Producer-Consumer

- one or more producer threads create some work and place it in a buffer or queue
- one or more consumer threads acquire that work from the queue and complete it
- queue between the producers and consumers is a bound resource
  - producers must wait for free space in the queue before writing
  - consumers must wait until there is something in the queue to consume
- coordination between the producers and consumers via the queue involves producers and consumers signaling each other
- producers write to the queue and signal that the queue is no longer empty
- consumers read from the queue and signal that the queue is no longer full
- both potentially wait to be notified when they can continue

Readers-Writers

- involves a shared resource that primarily serves as a source of information for readers but occasionally updated by writers
  - throughput is an issue
- emphasizing throughput can cause starvation and the accumulation of stale information
- allowing updates can impact throughput
- coordinating readers so they do not read something a writer is updating and vice versa requires coordination
  - writers tend to block many readers for a long period of time

    - need to balance the needs of both readers and writers to
        - satisfy correct operation
        - provide reasonable throughput
        - avoid starvation
- simple strategy
    - writers wait until there are no readers before allowing the writer to perform an update
    - if there are continuous readers writers will be starved
    - if there are frequent writers and they are given priority throughput will suffer
    - must find balance and avoid concurrent update issues

Dining Philosophers

- n actors (e.g. philosophers) in circular configuration
    - n means for accessing a resource (e.g. fork for food)
    - actors mostly wait (busy wait)
    - must possess 2 of the n means for accessing resource to access
    - when ready to access resource, actor takes control of 2 of the n means for accessing resource
    - if actor to right or left is already using one of n means, ready actor must wait
    - when done accessing resource, releases both of the reserved n means and waits
- actors ~= threads
- n means for accessing resource ~= locks/data/resources
- similar to many enterprise applications in which processes compete for resources
- systems that compete in this way can experience deadlock, livelock, throughput, and efficiency degradation

Beware Dependencies Between Synchronized Methods

- if there is more than one synchronized method on the same shared class then system may be written incorrectly
- recommendation
    - avoid using more than one method on a shared object
- there will be times when must use more than one method on a shared object
    - client-based locking
        - have the client lock the server before calling the first method and make sure the lock's extent includes code calling the last method
    - server-based locking
        - within the server create a method that locks the server, calls all the methods, and then unlocks
        - have the client call the new method

    - adapted server
        - create an intermediary that performs the locking
        - this is an example of server-based locking where the original server cannot be changed

Keep Synchronized Sections Small

- the synchronized keyword introduces an intrinsic lock
- all sections of code guarded by the same lock are guaranteed to have only one thread executing through them at any given time
    - locks are expensive because they create delays and add overhead
    - critical sections must be guarded
    - design code with as few critical sections as possible
- a critical section is any section of code that must be protected from simultaneous use for the program to be correct
- some naive programmers try to achieve this by making their critical sections very large
- extending synchronization beyond the minimal critical section increases contention and degrades performance
- recommendation
    - keep your synchronized sections as small as possible

Writing Correct Shut-Down Code Is Hard

- graceful shutdown can be hard to get correct
- common problems involve deadlock, with threads waiting for a signal to continue that never comes
    - parent waiting for child threads to shut down
    - dependent children waiting for each other (e.g. producer/consumer)
- recommendation
    - think about shut-down early and get it working early (takes longer than expected)
    - review existing algorithms because this is harder than expected

Testing Threaded Code

- proving correctness is impractical and testing does not guarantee it
- testing can minimize risk
- recommendations
    - write tests that have the potential to expose problems and then run them frequently
    - run with different programmatic configurations and system configurations and load
    - track down all test failures (don't ignore failures when there are subsequent successes)
    - treat spurious failures as candidate threading issues.
    - get nonthreaded code working first.
    - make threaded code pluggable
    - make threaded code tunable

    - run with more threads than processors
    - run on different platforms
    - instrument code to try and force failures

Treat Spurious Failures as Candidate Threading Issues

- threaded code causes things to fail that simply cannot fail
- most developers do not have an intuitive feel for how threading interacts with other code (authors included).
- bugs in threaded code might exhibit their symptoms once in a thousand, or a million, executions.
- best to assume that one-offs do not exist The longer these “one-offs” are ignored, the more code is built on top of a potentially faulty approach
- recommendation
    - do not ignore system failures as one-offs.

Get Your Nonthreaded Code Working First

- make sure code works outside of its use in threads
- recommendation
    - do not try to chase down nonthreading bugs and threading bugs at the same time
    - make sure code works outside of threads

Make Your Threaded Code Pluggable

- write concurrency-supporting code such that it can be run in several configurations
    - one thread, several threads, varied as it executes
    - threaded code interacts with something that can be both real or a test double
    - execute with test doubles that run quickly, slowly, variable
    - configure tests so they can run for a number of iterations
- recommendation
    - make thread-based code especially pluggable so that can be run in various configurations

Make Your Threaded Code Tunable

- getting the right balance of threads typically requires trial an error
- find ways (early on) to time the performance of your system under different configurations . Allow the number of threads to be easily tuned. Consider allowing it to change while the system is running. Consider allowing self-tuning based on throughput and system utilization.

Run with More Threads Than Processors

- context switches cause unexpected behavior
- run with more threads than processors or cores to encourage task swapping

- more frequently tasks swap, the more likely to encounter code that is missing a critical section or causes deadlock

Run on Different Platforms

- recommendation
    - run your threaded code on all target platforms early and often

Instrument Your Code to Try and Force Failures

- normal for bugs in concurrent code to hide indefinitely
- instrument code and force it to run in different orderings by adding calls to methods like Object.wait(), Object.sleep(), Object.yield() and Object.priority()
- hand-coded
    - may be only option
    - many problems with approach
        - manually located
        - difficult to find location to put calls to wait, sleep, etc
        - leaving such code in slows down production code
        - may or may not find results
- automated
    - many tools for doing this
- jiggle the code so that threads run in different orderings at different times
- the combination of well-written tests and jiggling can dramatically increase the chance of finding errors
- recommendation
    - use jiggling strategies to ferret out errors