CS3211 Notes

Intro

Concurrency: Possibility of multiple tasks starting, executing and completing in *overlapping* periods.

May or may not be running at same instance (e.g. Temporal Multithreading, single CPU context switching/interleaving multiple tasks), but they must all independently make progress.

Parallelism: Subset of concurrency, where tasks really do execute and make progress simultaneously.

Requires hardware-level multithreading (e.g. multi-core, Simultaneous Multithreading).

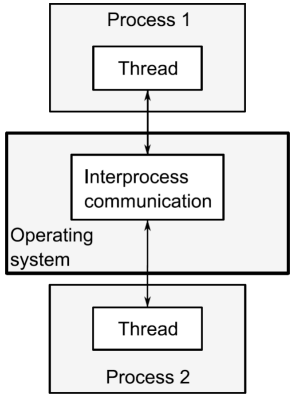
Exception: Disruption caused by machine level instruction (e.g. over/underflow, illegal memory access, memory misalignment).

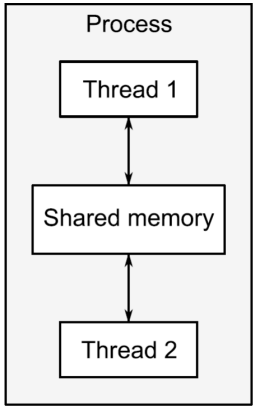
Synchronous - Happens due to program execution.

Interrupts: Disruption caused by external events (e.g. timeout, user input).

Asynchronous - Independent of program execution,

Threads vs Processes





Processes:

- Safe
- Easy to manage
- Costly creation
  - Syscall overhead
  - Data structures must be alloc-ed and copied
- Costly inter-proc communication via OS

Threads:

Extension of process model. Each process consist of multiple control flows called **threads**.

Each thread is a sequential execution stream within a process, with independent registers and stacks.

Threads share address space.

- Performant
- Cheaper to generate
  - No address space copying
- Easy to parallelize
  - OS can assign threads to different cores if they exist

Data Race - Syntax

1. Two concurrent control flows access a **shared resource**.
2. There is **no synchronization.**
3. At least one thread **modifies** the shared resource.

Race Condition - Semantics

1. When the outcome / correctness of a program depends on **timing or ordering** of events.
2. The events are executed independently and **concurrently** (e.g. on different threads).

One is not the subset, or determined by, the other. Concurrent accesses can race without compromising correctness and vice versa.

Race conditions need to be avoided by synchronizing access to data structures.

Critical Section

Requirements:

1. **Safety**

a. **Mutual Exclusion** - One thread of execution never enters a critical section while a concurrent thread of execution is already accessing it.

2. **Liveness**

   a. **Progress** - Threads not in the critical section cannot *prevent each other* from entering critical section. Threads in critical section will eventually leave.

   b. **No Starvation*** - Threads waiting on critical section will *eventually enter*.

3. **Performance** - Overhead of entering and exiting critical section should be small w.r.t work done.

Synchronization Mechanisms

1. **Locks** - Mutual exclusion primitive, used to build others.
2. **Condition Variables** - Wait for a condition to be true
3. **Binary Semaphores** - Generalized locks
4. **Counting Semaphores** - Limited number of threads can access.
5. **Monitors** - Lock + Condition Variables
6. **Barriers** - Wait for a group of threads to reach a point in code.
7. **Read-Write Locks** - Multiple readers, single writer
8. **Messages** - Transfer data as messages in atomic channels between threads

Deadlock

Occurs when two threads each holds a resource that the other needs, neither progressing.

Arises from competition for limited resources or incorrect synchronization.

Conditions

1. **Mutual Exclusion** - At least one resource must be held in a non-shareable mode.
2. **Hold and Wait** - A thread must hold a resource and be waiting to acquire additional resources.
3. **No Preemption** - A resource can only be released voluntarily by the thread holding it (no stealing or aborting externally).
4. **Circular Wait** - A set of threads must be waiting on each other in a circular chain.

See also: Dining Philosophers (https://en.wikipedia.org/wiki/Dining_philosophers_problem)

Approaches

1. **Prevention** - Ensure at least one of the four conditions cannot hold.
2. **Avoidance** - Control resource allocation.
3. **Detection** - Look for cycles in resource allocation graph and recover.

Starvation

Occurs when a thread is perpetually denied access to a resource.

1. **Unfairness** - Threads are not granted access in order of request.
2. **Priority Inversion** - Lower priority OS process holding a resource needed by a higher priority process is preempted by a medium priority process.

Why Concurrency?

Disadvantages:

1. **Complexity** - Harder to debug, reason about correctness.
2. **Performance** - Overhead of synchronization, thread creation, context switching.

Advantages:

1. **Separation of concerns** - E.g. GUI and backend.
2. **Performance** - E.g. Optimization, fully utilize hardware.

Types of Parallelism

1. **Task Parallelism** - Different operations on same data.

   Do same work in less time.
   E.g. Web server, Pipelining

2. **Data Parallelism** - Same operation on different data. (Embarrassingly parallel)

   No dependencies between data, no synchronization required.

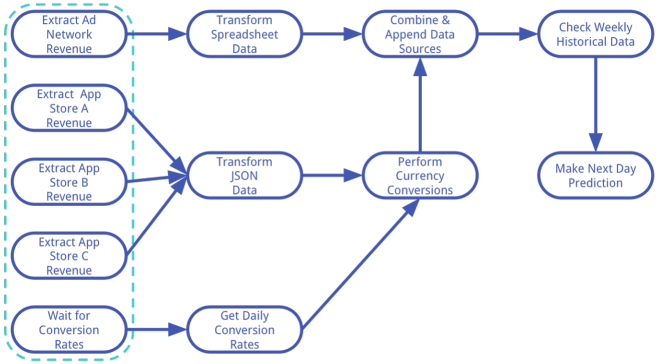
   Do more work in same time.
   E.g. Matrix multiplication, image processing, Monte Carlo simulations

Challenges

1. **Optimization** - How much can we parallelize? (Amdahl's Law)
2. **Granularity** - How much work to assign to each thread?
3. **Locality** - How to assign work to threads to minimize communication overhead?
4. **Coordination** - How to synchronize threads to ensure correctness?

5. **Debugging** - How to debug concurrent programs?
6. **Performance** - How to measure and monitor?

Task Dependency Graph



- Use to visualize task decomposition strategy.
- **Directed acyclic graph.**
  - Nodes - Tasks. Value - Execution time.
  - Edges - Control dependency
- Critical path length: maximum completion time.
- Degree of concurrency: Total Work / Critical Path Length.

Amdahl's Law

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

> Speedup of a program is limited by the fraction of the program that cannot be parallelized.

$0 \le f \le 1$ is known as the *sequential fraction* or *fixed-workload performance*.

$S_p(n) = T_*(n) f \times T_*(n) + {}_{1-fp} \times T_*(n) = 1f + {}_{1-fp} \le 1f$

Synchronization in C++

Threads

Creating Threads

```
#include <thread>

void foo(int x) {
    std::cout << x << std::endl;
}

int main() {
    std::thread t1(foo, 42); // create thread
    t1.join(); // wait for thread to finish
    return 0;
}
```

Using function objects:

```
...
class Foo {
    public:
        void operator()(int x) {
            std::cout << x << std::endl;
        }
};
...
    Foo foo;
    std::thread t1(foo, 42); // create thread
...
```

```
    std::thread t1(Foo(), 42);
    // this is wrong, it declares a function t1 returning a thread

    std::thread t1((Foo()), 42);
    std::thread t1{Foo(), 42};
    // these are correct, they declare a thread t1 running Foo() with argument 42
```

Using lambda:

```
    std::thread t1([](int x) {
        std::cout << x << std::endl;
    }, 42);
```

Wait vs Detach

```
t1.joinable(); // true
t1.join(); // wait for thread to finish
```

```
t1.detach(); // thread is now a daemon
t1.joinable(); // false
// thread is suspended when main() exits.
// stack is not unwound and destructors are not called,
// which may lock, corrupt or leak resources.
```

```
int main() {
    {
        int x = 42;
        std::thread t1(foo, x);
        t1.detach();
    }
    // lifetime of x ends here, but thread may still be running
}
```

Passing Arguments

Pass by Value

```
int x = 42;
std::thread t1(foo, x); // copy of x is passed to foo
```

```
void foo(std::string const& x);
void oops() {
    char buffer[1024];
    sprintf(buffer, "%s", "hello");
    std::thread t1(foo, buffer);
    t1.detach();
    // buffer passed by reference.
    // may not be converted to std::string in new thread.
}
```

```
void not_oops() {
    ...
    std::thread t1(foo, std::string(buffer));
    // buffer is converted before copy to new thread
    ...
}
```

Pass by Reference

```
void foo(data& x);
void oops(data x) {
    std::thread t1(foo, std::ref(x)); // pass x by reference
    ...
}
```

Ownership

```
std::thread t1(foo, 42);
std::thread t2 = std::move(t1); // t2 now owns t1
t1.joinable(); // false
t2.join(); // wait for thread to finish
```

RAII

Lifecycle of resource that must be acquired before use (allocated heap memory, file handle, mutex, network socket, etc.) is tied to an object's lifetime.

```
std::thread f {
    void some_function();
    return std::thread(some_function); // transfer ownership out of function
}
std::thread g() {
    void some_other_function(int);
    std::thread t(some_other_function, 42);
    return t;
}
```

```
void f(std::thread t);
void g() {
    void some_function();
    f(std::thread(some_function));// transfer ownership into function
    std::thread t(some_function);
    f(std::move(t)); // transfer ownership into function
}
```

Synchronization data writes

Mutex

```
std::mutex m;
{
    m.lock();
    // critical section
    m.unlock();
}
{
    std::lock_guard<std::mutex> lock(m); // RAII
    // critical section
}
{
    std::lock_guard lock{m}; // curly braces when omitting template arguments
    // or it will be parsed as func (lock) declaration
}
{
    std::unique_lock lock{m, std::defer_lock}; // RAII + defer lock
    // non-critical section
    lock.lock();
    // critical section
    lock.unlock();
    // non-critical section
    lock.lock();
    // critical section
}
```

```
std::mutex m2;
{
    std::scoped_lock lock{m, m2}; // RAII
    // critical section
} // Equivalent to:
{
    std::lock(m, m2);
    std::lock_guard<std::mutex> lock1(m, std::adopt_lock);
    std::lock_guard<std::mutex> lock2(m2, std::adopt_lock);
}
```

Conditional Variable

```
std::mutex m;
bool ready = false;
void bar() {
    std::unique_lock lock(m);
    ready = true;
    cv.notify_one(); // notify one thread waiting on cv
    // cv.notify_all(); // notify all threads waiting on cv
}
void foo() {
    std::unique_lock lock(m);
    cv.wait(lock, []{ return ready; }); // wait on cv until ready is true
    // do something
}
```

Spurious wake: At `cv.wait(...)`, thread may wake up and check condition even if not notified.

Memory Model in C++

Invariant: Memory ops should execute in order equivalent to some sequentially consistent interleaved execution in each thread.

Fact: Compiler/Processor may reorder instructions to optimize performance. E.g.

```
x = 0
for i=0...100:
    x = 1
    print x
```

becomes

```
x = 1
for i=0...100:
    print x
```

Problem: Reordering may break program semantics.

The C++ compiler is allowed to reorder instructions as long as:

1. Data written to files is the same as if the instructions were executed ‘as written’.
2. Prompts and waits for input are not reordered wrt interactive devices.

However, programs with undefined behavior are not constrained by the above rules.

Object Structure

In C++, objects occupy a contiguous region of memory, which can be one or more memory locations. Variables of fundamental types occupy *exactly* one memory location.

Modification Order

Total order of all modifications (writes) to an object, from all threads.

Need to ensure sequence of values for an object is consistent (not necessarily the same) across all threads.

If object is not atomic:

1. Programmer is responsible for ensuring threads agree on modification order (through synchronization).

If object is atomic:

1. Compiler is responsible for ensuring necessary synchronization.

Atomics

```
#inculde <atomics>

std::atomic<int> x = 0;

assert(x.is_lock_free());
// done by atomic instructions instead of internal mutex
assert(x.is_always_lock_free())
// ...across all platforms

x.store(42);
int y = x.load();
```

`std::memory_order` specifies how memory modifications are ordered around atomic operations.

Operation	Order		Operation	Order
Default	`..._seq_cst`		`load()`	`..._seq_cst`
`store()`	`..._seq_cst`,			`..._acquire`
	`..._release`			`..._relaxed`
	`..._relax`			
`compare_exchange...`	`..._seq_cst`,			
	`..._acq_rel`			
	`..._acquire`			
	`..._release`			
	`..._relaxed`			

Ordering relationships

1. Sequence-before: `a` sequences-before `b` if `a` occurs before `b` in the modification order of the same thread.

2. Synchronizes-with: `a` synchronizes-with `b` if `a` is a release operation and `b` is an acquire operation on the same object.

   Note, does not require `rel-acq` pair E.g. spinlock.

3. Inter-thread Happens-before
   `a` is inter-thread happens-before with `b` if any of the following are true

- `a` synchronizes with `b`
- `a` is dependency-ordered before `b` (not explained)
- `a` synchronizes with `x` and `x` is sequenced-before `b`
- `a` is sequenced-before `x` and `x` is inter-thread happens-before `b`
- `a` inter-thread happens-before `x` and `x` is inter-thread happens-before `b` (transitivity)

1. Happens-before: `a` happens-before `b` if:

   1. `a` sequences-before `b`.
   2. `a` synchronizes-with `b`. (Inter-thread happens-before)
   3. There exists an operation `c` such that `a` happens-before `c` and `c` happens-

before `b`.

2. Visible side-effect: A side-effect A is visible to B if

    1. A happens-before B.
    2. No side-effect C 'occurs-between' A and B.

    The visible side-effects of B are the side effects of all operations that **B does not happen-before**.

Sequentially Consistent Ordering

- All threads see the same ordering of operations on all variables.
- Operations are not reordered.
- Stores *synchronize-with* loads on the same variable.
- Requires extensive synchronization operations on weakly ordered architectures.

```
std::atomic<int> x = 0;
std::atomic<int> y = 0;
std::atomic<int> z = 0;

void write_x() {
    x.store(1, std::memory_order_seq_cst);
}
void write_y() {
    y.store(1, std::memory_order_seq_cst);
}
void read_x_then_y() {
    while (x.load(std::memory_order_seq_cst) == 0);
    if (y.load(std::memory_order_seq_cst) == 1) {
        ++z;
    }
}
void read_y_then_x() {
    while (y.load(std::memory_order_seq_cst) == 0);
    if (x.load(std::memory_order_seq_cst) == 1) {
        ++z;
    }
}
```

```
int main() {
    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);
    // true, either write_x happens-before write_y or vice versa.
}
```

Relaxed Ordering

Operations are allowed to be 'delayed' but 'progressive' wrt a single thread.

Once a thread has read a value for an object, it must:

1. see same or later values on subsequent reads.
2. have writes to that object occur later in its modification order.

Once a thread has written a value for an object, it must:

1. have the write occur at the bottom of its modification order.
2. see the same or later values on subsequent reads.

Between two objects, threads may see different interleaving of modifications, as long as the modification order of *each* object is still consistent.

- Operations performed with relaxed ordering do not participate in synchronizes-with relationships.

- Operations to the same variable cannot be reordered and still obey happens-before relationships.

In example above, due to reordering of writes, `thread c(read_x_then_y)` may see `x=1,y=0` while `thread d(read_y_then_x)` sees `x=0,y=1`. The assert can be false.

Release-Acquire Ordering

- Release operations synchronize-with acquire operations on the same variable.
- **All writes that happened-before a release operation** in a thread are **visible side effects** to all threads that perform an acquire operation on the same variable.
- Acquires can 'synchronize-with' initialization as well. If there are multiple releases in

different threads, the synchronization order is arbitrary.

```
std::atomic<bool> x, y;
std::atomic<int> z;

void write_x_then_y() {
    x.store(true, std::memory_order_relaxed);
    y.store(true, std::memory_order_release);
}
void read_y_then_x() {
    while (!y.load(std::memory_order_acquire));
    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);
    // true, y.load synchronizes-with y.store happens-before x.load.
    // would be false if ops on y were relaxed.
    // can use std::atomic_thread_fence.
}
```

Language-level Memory Model

Modern languages guarantee sequential consistency for data-race-free programs (SC for DRF)

- Compilers insert synchronization operations to cope with hardware memory model. (E.g. relaxed / seq-rel on atomics)

Debugging

Unwanted Blocking

1. **Deadlock**
2. **Livelock** - States of threads constantly changing, but no progress is made. Can fool deadlock detection algorithms.
3. **Blocking on I/O** - Threads waiting for external input.

Race Conditions

1. **Data Races** - Lack of / improper synchronization.

2. **Broken Invariants**
    - Dangling pointers / Use-after-free - thread deletes data being accessed
    - Double free - threads pop and delete same value from queue
    - Random memory corruption - thread throws exception while modifying data

3. **Lifetime issues**
    - Thread outlives data accessed
    - Local variable referenced by thread goes out of scope.
    - `join()` not called in exception path

Tools

- **Valgrind**
    - Binary instrumentation for shadowing
    - Logs all registers and memory accesses
    - Thread error detectors: Helgrind, DRD
- Sanitizer tools
    - Compiler-based (`-fsanitize=thread`)

Need to keep track of memory state and compute happens-before.

Data Structures

Goal:

- Support multiple concurrent access to underlying data
- Each thread sees a self-consistent view of data
- Thread-safety
    - No data loss or corruption
    - All invariants upheld
    - No race conditions

Options:

- Lock-based vs lock-free
- Fine-grained vs coarse-grained synchronization

High-level mutex not practical as **serialization** prevents true concurrent access to data.

Course-grained Queue

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

    std::shared_ptr<T> try_pop() {
        std::lock_guard<std::mutex> lk(mut);
        if (data_queue.empty()) {
            return std::shared_ptr<T>();
        }
        std::shared_ptr<T> res = data_queue.front();
        data_queue.pop();
        return res;
    }



    std::shared_ptr<T> wait_and_pop() {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this]{return !data_queue.empty();});
        std::shared_ptr<T> res = data_queue.front();
        data_queue.pop();
        return res;
    }

    void push(T new_value) {
        std::shared_ptr<T> data(std::make_shared<T>(std::move(new_value)));
        // alloc done outside lock to improve performance
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }

    bool empty() const {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

Fine-grained Queue

Pre-allocate with dummy node to:

1. Separate front node from back
2. Prevent race on `front->next` and `back->next`

- When queue is empty, `front` and `back` point to dummy node.
- When pushing, we **modify dummy node with new data** and point it and `back` to new dummy node.
    - If queue is empty, this overwrites front as well. Thus both locks are needed.

```
template<typename T>
class threadsafe_queue {
private:
    struct node {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next; // enable implicit delete on data when not used.
    };
    std::mutex front_mutex;
    std::unique_ptr<node> front;
    std::mutex back_mutex;
    node* back;
    node* get_back() {
        std::lock_guard lk{back_mutex};
        return back;
    }
    std::unique_ptr<node> pop_front() {
        std::lock_guard lk{front_mutex};
        if (front.get() == get_back()) {
            return nullptr;
            // don't alloc empty std::shared_ptr here as it's expensive
        }
        std::unique_ptr<node> old_front = std::move(front);
        front = std::move(old_front->next);
        return old_front;
    }
```

```
public:
    threadsafe_queue() : front(new node), back(front.get()) {}
    threadsafe_queue(const threadsafe_queue& other) = delete;
    threadsafe_queue& operator=(const threadsafe_queue& other) = delete;

    std::shared_ptr<T> try_pop() {
        std::unique_ptr<node> old_front = pop_front();
        return old_front ? old_front->data : std::shared_ptr<T>();
    }

    void push(T new_value) {
        std::shared_ptr<T> new_data(std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        node* const new_back = p.get();
        std::lock_guard lk{back_mutex};
        back->data = new_data;
        back->next = std::move(p);
        back = new_back;
    }
};
```

Non-blocking data structures

- **Obstruction-free** - with all other threads are paused, one thread will complete in a finite number of steps.
- **Lock-free** - with multiple threads operating concurrently, at least one will complete in a finite number of steps.

Threads need to be able to access data concurrently, but not necessarily in the same operation.

E.g. lock-free queue allows concurrent `push` with `pop`, but not `push` with `push`.

Lock-free algorithms in loops (cmp_exchange) can result in starvation.

Lock-free algorithms can have livelocks e.g. when forced to retry.

- **Wait-free** - every thread operating concurrently will complete in a finite number of steps.

Spin-locking is not wait-free as it can be starved.

ABA problem

1. Thread 1 reads value A
2. Thread 2 preempts 1
3. Thread 2 writes value B
4. Thread 2 writes value A
5. Thread 1 resumes and compares value A, which is the same as before, so it thinks nothing has changed.

In a lock-free list, if an item is removed, delete, and a new one allocated and added, the new item may have the same address as the old one due to MRU memory allocation (caching). With compare-exchange, this can cause use-after-free.

Concurrency in Go

Communicating Sequential Processes (CSP)

- Break program into independent concurrent processes
- Communicating processes via channels

```
func push(wg *sync.WaitGroup, j int, c chan<- int) {
    defer wg.Done()
    c <- j
}()

func main() {
    var wg sync.WaitGroup
    ct := make(chan time.Time, 4)
    for i, j := range []int{4,3,2,1} {
        c := make(chan int)
        wg.Add(2)
        go push(&wg, c, j)
        go func(i int) {
            res, ok := <-c
            fmt.Printf("func %d got %d at", i, res)
            // runtime transfers memory of c, ct to heap
            ct <- time.Now()
            wg.Done()
        }(i) // pass i by copy
    }
    wg.Add(1)
    go func() {
        defer wg.Done()
        for t := range ct {
            fmt.Printf("completion at %v", t)
        }
    }()
    wg.Wait()
}
```

Goroutines

Cheaper than threads, and run in the same address space as the main program.

- Runtime *multiplexes* goroutines onto OS threads.
  - Automatic scheduling - mapping M:N
  - **Decoupling concurrency from parallelism**
- Blocking goroutine blocks the underlying OS thread, but not other goroutines.

Not garbage collected (programmer responsibility to sync and prevent leakage).

Channels

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

Ownership

Owner should...	**Consumer should...**
Instantiate channel	Check if channel is closed
Perform writes	Responsibly handle blocking for any reason
Pass ownership to another goroutine	
Close channel	
Encapsulate above and expose via reader channel	

In unidirectional channels,

- Owners have write-access view of channel (`chan` or `chan<-`)
- Utilizers only have read-access view of channel (`<-chan`)

Having the owner initialize the channel removes the risk of...

- deadlocking by writing to a nil channel
- panicking by closing a nil channel

Having the owner close the channel removes the risk of...

- panicking by writing to a closed channel
- panicking by closing a closed channel

Select statement

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

For-select loop:

```
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\n", workCounter)
```

Memory model

- `go` statement that starts a goroutine is **synchronized-before** its execution begins.
- The completion of the goroutine is **NOT** synchronized-before any event in the program.
- A send on a channel is **synchronized-before** the corresponding `receive` from that channel completes.
- The closing of a channel is **synchronized-before** a zero-value receive.
- A receive from an un-buffered channel is **synchronized-before** the send on that channel completes.
- The kth receive on a channel with capacity C is **synchronized-before** the (k+C)th send from that channel completes.

Confinement

- **Ad-hoc confinement** - programmer responsibility to ensure that data is modified only by a single goroutine.
- **Lexical confinement** - restricting access to shared locations.

1. By exposing only the reader/writer handle.

```
consumer := func(results <-chan int){
    ...
```

2. By exposing only slices of data

```
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()
```

Pipelining

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

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

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

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

Fan-out, fan-in

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

```
    // Select from all the channels
    wg.Add(len(channels))
    for _, c := range channels {
        go multiplex(c)
    }
    // Wait for all the reads to complete
    go func() {
        wg.Wait()
        close(multiplexedStream)
    }()
    return multiplexedStream
}

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

Classic synchronization problems

Producer-consumer (buffered)

Producer	Consumer

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

Readers-writers (Lightswitch)

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

Writers	Readers

`roomEmpty.wait();`
`write();`
`roomEmpty.signal();`
	`readLightswitch.lock(roomEmpty);`
	`read();`
	`readLightswitch.unlock(roomEmpty);`

To prevent starvation, use turnstile = Semaphore(1).

Writers	Readers

`turnstile.wait();`
`roomEmpty.wait();`
`write();`
`turnstile.signal();`
`roomEmpty.signal();`
	`turnstile.wait();`
	`turnstile.signal();`
	`readLightswitch.lock(roomEmpty);`
	`read();`
	`readLightswitch.unlock(roomEmpty);`

Readers-writers lock

```
mutable std::shared_mutex mutex_
...
std::shared_lock lock(mutex_);
...
std::unique_lock lock(mutex_);
```

Barrier

```cpp
int main() {
    std::barrier barrier(2);
    std::thread t1([&barrier] {
        std::cout << "t1 before barrier\n";
        barrier.arrive_and_wait();
        std::cout << "t1 after barrier\n";
    });
    std::thread t2([&barrier] {
        std::cout << "t2 before barrier\n";
        barrier.arrive_and_wait();
        std::cout << "t2 after barrier\n";
    });
    t1.join();
    t2.join();
}
```

```cpp
struct Barrier {
    std::ptrdiff_t expected;
    std::ptrdiff_t count;
    std::mutex mut;
    std::counting_semaphore<> turnstile;
    std::counting_semaphore<> turnstile2;

    Barrier(std::ptrdiff_t n) :
        expected(n), count(0), mut{}, turnstile(0), turnstile2(1) {}

    void arrive_and_wait() {
        {
            std::scoped_lock lock(mut);
            if (++count == expected) {
                turnstile2.acquire();
                turnstile.release(expected);
            }
        }
        turnstile.acquire();
        {
            std::scoped_lock lock(mut);
            if (--count == 0) {
                turnstile.acquire(); // prevent threads from lapping the barrier
                turnstile2.release(expected);
            }
        }
        turnstile2.acquire();
    }
}
```

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

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

func (b *Barrier) ArriveAndWait() {
    b.wg.Done()
    b.wg.Wait()

    b.wg.Add(1)
    b.wg2.Done()

    b.wg2.Wait()
    b.wg2.Add(1)
}
```

Rust

Ownership

- Giving ownership is the default. Like `std::move` in C++, but enforced at compile time without retention.

- Deep copy of data is explicit using `clone`.

```rust
fn publish(book: Vec<String>) {
...
let mut book = Vec::new();
book.push("A Dance With Dragons");
publish(book); // ownership is moved to publish
println!("{} is no longer available", book[0]); // error: use of moved value
```

```rust
publish(&book); // borrow book reference
...
fn publish(book: &Vec<String>) {
    book.push("A Dance With Dragons"); // error: mutating borrowed content
```

```
publish(&mut book); // borrow book reference mutably
...
fn publish(book: &mut Vec<String>) {
    book.push("A Dance With Dragons"); // ok
```

Borrows are scoped:

```
let mut book = Vec::new();
book.push("A Dance With Dragons");
{
    let borrow1 = &book;
    book.push("A Feast for Crows"); // error: cannot mutate borrow
    borrow1.push("A Clash of Kings"); // error
}
book.push("A Storm of Swords"); // ok
```

```
let mut book = Vec::new();
book.push("A Dance With Dragons");
{
    let borrow2 = &mut book;
    book.len(); // error: cannot access while mutable borrow exists
    let borrow1 = &mut book; // error: cannot mutably borrow more than once
    borrow2.push("A Clash of Kings"); // ok
}
book.push("A Storm of Swords"); // ok
```

Safety

Type	Ownership	Alias?	Mutate?
T	Owned	No	Yes
&T	Shared reference	Yes	No
&mut T	Mutable reference	Yes	Yes

- Borrow-checker *statically* prevents **aliasing + mutation**
- Ownership prevents **double-free**
  - Owner frees
- Borrowing prevents **use-after-free**
- Data race = sharing + mutation + no ordering
  - First two are prevented in rust

Threading

```
let loc = thread::spawn(|| {
    "world"
});
println!("hello, {}", loc.join().unwrap());
```

```
let mut dst = Vec::new();
thread::spawn(move || {
    // move makes closure take ownership of dst.
    //Otherwise, it would be borrowed.
    dst.push(1); // compiler can infer whether capture should be mutable or not
});
dst.push(2); // error: use of moved value
```

Error if there were no `move`, as `dst` may not live long enough before the thread starts.

Traits

Similar to interface, but can have default implementations.

```
trait Clone {
    fn clone(&self) -> Self;
}
impl<T: Clone> Clone for Vec<T> {
    fn clone(&self) -> Vec<T> {
        let mut v = Vec::new();
        for e in self {
            v.push(e.clone());
        }
        v
    }
}
```

Atomic Reference Counting

Only allows shared (immutable) references. Use with mutex to utilize interior mutability.

```
let v = Arc::new(Mutex::new(Vec::new()));
let mut handles = Vec::new();
for i in 0..10 {
    let v = v.clone();
    handles.push(thread::spawn(move || {
        let mut v = v.lock().unwrap();
        v.push(i); // interior mutability
    }));
}
for handle in handles {
    handle.join().unwrap();
}
println!("{:?}", v.lock().unwrap());
```

Atomics

```
let x = AtomicUsize::new(0);
let prev = x.fetch_add(1, Ordering::SeqCst);
assert_eq!(prev, 0);
let prev = x.swap(10, Ordering::SeqCst);
assert_eq!(prev, 1);
assert_eq!(x.load(Ordering::SeqCst), 10);
```

Channels

MPSC FIFO queue.

```
let (tx, rx) = mpsc::channel();
let tx2 = tx.clone();
thread::spawn(move || {
    tx.send(1).unwrap();
});
thread::spawn(move || {
    tx2.send(2).unwrap();
});
for _ in 0..2 {
    println!("{}", rx.recv().unwrap()); // prints 1 and 2 in any order
}
```

Scoped Threads

- Cannot borrow mutably into two threads in same scope
- MPMC channels with exponential backoff (from Crossbeam)

```
fn main() {
    let mut v = vec![0; 10];
    println!("main has id {}", thread_id::get());

    std::thread::scope(|s| {
        s.spawn(|_| {
            println!("thread spawned with id {}", thread_id::get());
            v[0] = 1;
        });
    }).unwrap();

    println!("v[0] = {}", v[0]);
}
```

```
fn main() {
    let (sender, receiver) = bounded(CAPACITY);

    let mut handles = Vec::new();
    for _ in 0..NTHREADS {
        let sender = sender.clone();
        handles.push(thread::spawn(move || {
            let mut rng = rand::thread_rng();
            let mut backoff = Backoff::new();
            for _ in 0..NITEMS {
                while sender.is_full() {
                    backoff.snooze();
                }
                sender.send(rng.gen()).unwrap();
            }
        }));
    }
```

```
    for _ in 0..NTHREADS {
        let receiver = receiver.clone();
        handles.push(thread::spawn(move || {
            let mut backoff = Backoff::new();
            for _ in 0..NITEMS {
                while receiver.is_empty() {
                    backoff.snooze();
                }
                let item = receiver.recv().unwrap();
                println!("got {}", item);
            }
        }));
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

```
fn fetch_mul(a: &AtomicUsize, b: usize) -> usize {
    let backoff = Backoff::new();
    loop {
        let val = a.load(Ordering::SeqCst);
        if a.compare_and_swap(val, val * b, Ordering::SeqCst) == val {
            return val * b;
        }
        backoff.spin();
    }
}
```

Data Parallelism

with Rayon

```
fn main() {
    let mut v = init_vec();
    let max = AtomicUsize::new(MIN);
    v.par_iter().for_each(|&x| {
        loop {
            let old_max = max.load(Ordering::SeqCst);
            if x <= old_max {
                break;
            }
            let new_max = max.compare_and_swap(old_max, x, Ordering::SeqCst);
            if new_max == old_max {
                break;
            }
        }
    });
    println!("max = {}", max.load(Ordering::SeqCst));
}
```

Async/Await

- Non-blocking I/O allows single thread to do useful work while waiting.

- Futures help keep track of in-progress operations and associated state.
    - Represents a value that may not be ready yet.
    - Event loop polls futures until they are ready.

Zero-cost abstractions:

- Code you cannot write better by hand. E.g. iterators API.
- Futures.rs (http://Futures.rs) is zero-cost.

1. Executor calls `poll()` on future, passing in `Context` struct.
2. Future runs until no progress can be made.
3. Future returns `Poll::Ready(T)`, or `Poll::Pending` if it is waiting on some event.
4. Future calls `Context::wake()` when new progress can be made.
5. Executor uses `Context` to see which futures are ready to be polled again.

Futures **cannot** block, or the executor blocks.

Futures

```
fn addToInbox(email_id: u64, recipient_id: u64)
    -> impl Future<Output = Result<(), Error>>
{
    loadMessage(email_id)
        .and_then(|message| loadRecipient(recipient_id)
        .and_then(|(message, recipient)| recipient.verifyHasSpace(&message))
        .and_then(|(message, recipient)| recipient.addToInbox(&message))
}
```

async/.await

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

- Async functions return a future.
- They have no stack. All state is contained in the future.
- No recursion. Futures need to have a fixed size known at compile time.

Exam General Advice

Atomics

Can use atomics in a spin-wait manner to mimic locks, so data races are still possible with atomics.

Concurrency

1. Concurrency let's us use the power of parallelism in modern multi-core machines.
2. Concurrency is harder to reason about
3. Concurrency increases coding complexity
4. Concurrency might have higher memory usage

Debugging

1. Quantify effect by measing execution time with and without concurrency
2. Incrementally remove concurrency
3. Reduce problem size to ease debugging
4. Run single-threaded first
5. Use Valgrind/Helgrind/TSan/Asan/gdb

Language models for concurrency

Nowadays, Go's share-by-communicating ideology avoids a lot of synchronization problems, and Rust's Send+Sync+borrow checker enables fearless concurrency, so the language models have indeed progressed.

Language models for concurrency

Nowadays, Go's share-by-communicating ideology avoids a lot of synchronization problems, and Rust's Send+Sync+borrow checker enables fearless concurrency, so the language models have indeed progressed.

Race Conditions

Can still occur without data races, by having timing/ordering of events affecting correctness.

Can use ato

- Allows for easy distributed systems
- Data is protected by confinement, rather than synchronization
- Clear ownership / SoC
- Lower performance / loses locality
- More tedious for some operations (updating a complex data structure)

Data structure

- just say data structure is cache friendly