Concurrency in Go

(or, Erlang done right)

Programming Model

Mostly CSP/π-calaculus (not actually formalized): goroutines+channels

Go concurrency motto:
"Do not communicate by sharing memory; instead, share memory by communicating"

+ Full-fledged shared memory

Goroutines

Think **threads** (however no *join* operation)

```
go foo()

go logger.Printf("Hello, %s!", who)

go func() {
    logger.Printf("Hello, %s!", who)
    ...
}()
```

Channels

Basically typed bounded blocking FIFO queues.

```
// synchronous chan of ints
c := make(chan int)

// buffered chan of pointers to Request
c := make(chan *Request, 100)
```

Channels: First-class citizens

You can pass them as function arguments, store in containers, pass via channels, etc.

Moreover, channels are not tied to goroutines. Several goroutines can send/recv from a single channel.

Channels: input/output

```
func foo(c chan int) {
  c <- 0
  <-c
}

func bar(c <-chan int) {
  <-c
}

func baz(c chan<- int) {
  c <- 0
}
```

Channels: closing

```
func producer(c chan *Work) {
   defer close(c)
   for {
       work, ok := getWork()
       if !ok { return }
       c <- work
    }
}
```

```
// consumer
for msg := range c {
   process(msg)
}
```

Why no goroutine join?

Usually it's required to return some result anyway.

```
c := make(chan int, N)

// fork
for i := 0; i < N; i++ {
  go func() {
    result := ...
    c <- result
  }()
}

// join
sum := 0
for i := 0; i < N; i++ {
  sum += <-c
}
```

Select

Select makes a pseudo-random choice which of a set of possible communications will proceed:

```
select {
  case c1 <- foo:
  case m := <-c2:
    doSomething(m)
  case m := <-c3:
    doSomethingElse(m)
  default:
    doDefault()
}
```

Select: non-blocking send/recv

```
reqChan := make(chan *Request, 100)

httpReq := parse()
select {
  case reqChan <- httpReq:
  default:
    reply(httpReq, 503)
}
```

Select: timeouts

```
select {
  case c <- foo:
  case <-time.After(1e9):
}
```

Example: Barber Shop

```
var seats = make(chan Customer, 4)

func barber() {
  for {
    c := <-seats
    // shave c
  }
}
go barber()

func (c Customer) shave() {
  select {
  case seats <- c:
  default:
  }
}
```

It is that simple!

Example: Resource Pooling

Q: General pooling functionality
Any has in their code somewhere, a good pooling mechanism for interface type maybe? Or just somthing one could adapt and abstract? I'm sure some db drivers should have this, any ideas?

Sounds a bit frightening...

Example: Resource Pooling (cont)

The solution is just

pool := make(chan *Resource, 100)

Put with [nonblocking] send.
Get with [nonblocking] recv.

It is *that* simple!

Haa, works like a charm.
Go makes things so nice and simple, yet so powerful!
Thanks

Example: Actor-oriented programming

```
type ReadReq struct {
  key string
  ack chan<- string
}

type WriteReq struct {
  key, val string
}

c := make(chan interface{})

go func() {
  m := make(map[string]string)
  for {
    switch r := (<-c).(type) {
    case ReadReq:
      r.ack <- m[r.key]
    case WriteReq:
      m[r.key] = r.val
    }
  }
}()
```

Example: Actor-oriented programming

c <- **WriteReq**{"foo", "bar"}

ack := make(chan string)
c <- **ReadReq**{"foo", ack}
fmt.Printf("Got", <-ack)

It is **that** simple!

Example: Thread Pool

[This page intentionally left blank]

Why does pure CSP suck?

CSP-style memory allocation:

```
ack1 := make(chan *byte)
Malloc <- AllocReq{10, ack1}

ack2 := make(chan *byte)
Malloc <- AllocReq{20, ack2}

obj1 := <-ack1
obj2 := <-ack2
```

WTF??1!

Why does pure CSP suck?

Some application code is no different!

```
var UidReq = make(chan chan uint64)
go func() {
  seq := uint64(0)
  for {
    c := <-UidReq
    c <- seq
    seq++
  }
}()

ack := make(chan uint64)
UidReq <- ack
uid := <-ack
```

Why does pure CSP suck?

“Message passing is easy to implement. But everything gets turned into distributed programming then” (c) Joseph Seigh

- Additional overheads
- Additional latency
- Unnecessary complexity (asynchrony, reorderings)
- Load balancing
- Overload control
- Hard to debug

Shared Memory to the Rescue!

```
var seq = uint64(0)
...
uid := atomic.AddUint64(&seq, 1)
```

Simple, fast, no additional latency, no bottlenecks, no overloads.

Shared Memory Primitives

sync.Mutex
sync.RWMutex
sync.Cond
sync.Once
sync.WaitGroup

runtime.Semacquire/Semrelease

atomic.CompareAndSwap/Add/Load

Mutexes

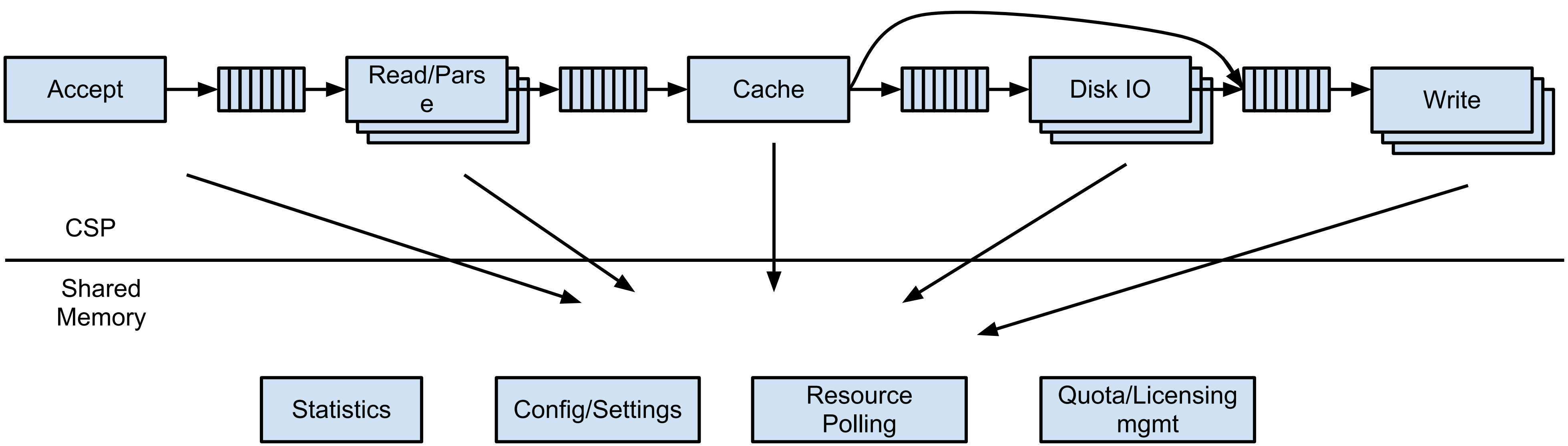
sync.Mutex is actually a cooperative binary semaphore - no ownership, no recursion.

```
mtx.Lock()
go func() {
    ...
    mtx.Unlock()
}
mtx.Lock()

func foo() {
  mtx.Lock()
  defer mtx.Unlock()
  ...
}
```

General Scheme

90% of CSP on higher levels
+10% of Shared Memory on lower levels



Race Detector for Go

Currently under development in Google MSK (no obligations to deliver at this stage).

The idea is to provide general support for dynamic analysis tools in Go compiler/runtime/libraries.

And then attach almighty ThreadSanitizer technology to it.

If/when committed to main branch, user just specifies a single compiler flag to enable it.

Scalability

The goal of Go is to support scalable fine-grained concurrency. But it is [was] not yet there.

Submitted about 50 scalability-related CLs. Rewrote all sync primitives (mutex, semaphore, once, etc), improve scalability of chan/select, memory allocation, stack mgmt, etc.

"I've just run a real world benchmark provided by someone using mgo with the r59 release, and it took about 5 seconds out of 20, without any changes in the code."

Still to improve goroutine scheduler.

Thanks!