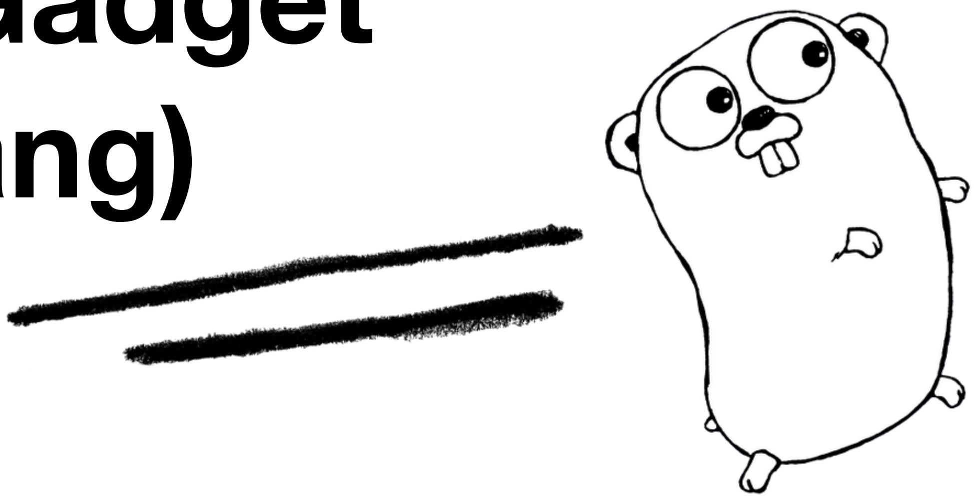
Go Go Gadget Go(lang)



Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

About This Lecture...

- I don't expect you to program in Go for this class...
  - But I admit I'm mightily tempted to encourage 61b to be redone in Go...
  - Or create a 161 project in Go
  - And I'd like to hope at least 10%+ of the class go "cool" and look into this in more detail
- But the concepts are very interesting
  - Interesting enough that I'm kicking myself for not starting to play with it much earlier:
    I only started playing with it on the first day of Spring
- In particular, its focus on concurrency maps well to multicore/multiprocessor systems
  - Requires shared memory, but programming without shared memory is a much bigger PitA and only works for some problems
  - Since the future is parallel, this is probably a language you should learn
- Just because this wont be on the test doesn't mean you shouldn't know it!

What is Go

- Language created at Google starting in 2007
  - Primarily by a bunch of old Unix hands: Robert Griesemer, Rob Pike, and Ken Thompson
  - 1.0 released in March 2012
- Language continues to evolve, but a commitment to backwards compatibility (so far)
  - A correct program written today will still work tomorrow
    - I'm looking at you, ***python 3....***
- Mostly C-ish but...
  - Strong typing, no pointer arithmetic, lambdas, interfaces, and...
  - Strong emphasis on concurrent computation

Concurrency and Parallelism

- ***Concurrency*** represents the ability to perform multiple things at the same time
- Reminder
  - SIMD: Single Instruction, Multiple Data
    - A GPU
  - Shared Memory MIMD: Multiple instruction, multiple data, common memory
    - A multicore processor
  - Clusters: Each computational group has its own independent memory

Concurrency doesn't always give parallelism

- Go is concurrent and supports parallel execution
  - The runtime can schedule multiple concurrent routines on separate CPU threads at the same time
    - So a concurrent program in Go running on a 4 core, 2 thread/core Intel processor can be running up to 8 separate streams of execution at one time
- Python's threading is concurrent ***but not parallel***:
  - Python has a "global interpreter lock":
    Can only execute a single thread of python bytecode at a time
  - Python threading code is good for waiting on I/O or special C libraries that release the global lock
  - But generally ***can not use multiple processors efficiently***

Good Go Resources

- The Go website:
  - https://golang.org/
- Especially useful: Effective Go:
  - A cheatsheet of programming idioms. Several example from this lecture stolen from there
  - https://golang.org/doc/effective_go.html
- When searching Google, ask for ***golang***, not go
  - The language may be Go, but golang refers to the language too

So Think of Go as:

- C's general structure & concepts
  - But with implied ;s and a garbage collector
- A better typing system with interfaces, slices, and maps
  - No class inheritance, however
- Much more symmetric functions
  - Can return multiple values
- Scheme-like lexical scope
  - Lambdas and interior function declarations
- Communicating Synchronous Processes (CSP) concurrency
  - Multiple things at once in the same shared memory space: quite suitable for MIMD

Go Typing System

- Go is statically typed with no automatic coercion
  - `var x int = 32 // Int is architecture dependent`
    `var y int64 ...`
  - `y = x          // This is an error`
  - `y = int64(x)   // this is correct`
- But it does have a lot of automatic type inference and creation with `:=`
  - `z := foo(...)        // z is created, type is return type of foo`
  - `a, b, _ := bar(...) // Bar returns 3 values, 3rd is ignored here`
- And it also has structures & pointers with automatic initialization to default values
  - ```
    type SyncedBuffer struct {
        lock    sync.Mutex
        buffer  bytes.Buffer
    }
    ```
  - `p := new(SyncedBuffer)  // type *SyncedBuffer`
  - `var v SyncedBuffer      // type  SyncedBuffer`
    ```

Go Memory Allocation

- Go is garbage collected (like Java)
- Its OK to return pointers to items on the stack
  - Allocator will keep that memory alive
  - ```
    ...
    foo := File{fd, name, nil, 0}
    return &foo
    ```
- But you may need to explicitly allocate memory
  - `new(T) // Creates a pointer to a zeroed object of type T`
  - `make(T, args) // Only creates channels, slices, and maps`
    ```

Slices and Arrays

- Arrays need to be statically declared
  - `foo string[5]`
- But slices are like python lists
  - But they are a view into an arrays
  - ```
    bar := foo[2:]
    bar[0] = "this_will_set foo[2] as well"
    ```
- Utilities to append by copying
  - bar := append(bar, "all", "this", "stuff")
    - Creates a new slice and, if necessary, copies the memory
    - Allows efficient manipulations
    ```

Defer

- A common motif in programming
  - Open a file/grab a lock/etc…
    - Do a bunch of stuff
  - (forget to) close the file/release the lock/etc…
- Defer allows you to delay the function execution to later
  - **`defer foo(a, b, c())`**
    - All arguments are processed (so c is executed immediately)
    - defer invokes the deferred functions on exit in last-in, first-out manner when the function exits completely

Functions

- Not only can you declare functions normally
  - `func foo(a string, b int)(c int, d error){…}`
- You can also declare them in a function body
  - Where they have scheme-like lexical scope
- And even declare autonomous functions
  - `defer func() {…} ()`
  - Will execute the function as a defer
- Has full scheme-like lexical scope
  - So you can access enclosing variables

Coroutines, err, goroutine

- Conceptually, a goroutine is just a thread...
  - `go fubar() // Executes fubar as a new coroutine`
- But in practice it is designed to be much lighter weight
  - Threads (e.g. in OpenMP) relatively expensive to create:
    Operating System involvement is never cheap
- Go's runtime instead pre-creates a series of threads
  - And then schedules the active coroutines itself to the available threads
- Result is goroutines are ***cheap***
  - It is only slightly more expensive than a plain function call:
    new goroutine just has a small independent stack
    context switching between goroutines is very cheap

Channels

- Channels are the primary synchronization mechanism
  - A typed and (optionally) buffered communication channel
  - ```
    c := make(chan int)
    e := make(chan fubar, 100)
    ```
- Writing data to a channel:
  - `c <- 32 // Writing blocks if unbuffered or full`
- Reading from a channel:
  - `var f = <- e // Reading blocks if no data`
    ```

Channels as Synchronization Barriers

- Any writes in the code before writing to the channel complete first
  - ```
    globalA = ...
    d <- 1   // The write to a must complete just before this
    ```
- Any reads in the code after reading from the channel do not start until channel-read takes place
  - ```
    <- d
    fubar(globalA) // Won't read globalA before channel read
    ```
- Otherwise, compiler can reorder however it wants as long as sequential semantics are preserved for the sequential function
  - Go may have removed a lot of ways to shoot yourself in the foot...
    but unless you use channels etc, you can easily blow it off with race conditions

Without Synchronization Barriers, The Compiler Can Go To Town

- This doesn't work!
  - Compiler can reorder the writes between x and done safely
    - Similar variants also possible
- This is one of the two biggest pitfalls of Go:
  - Unless you explicitly synchronize, multiple processes can write in "weird" ways
  - The other is the abysmal error/exception handling mechanism

```
var x : string
var done : bool
func foo(){
   ...
   x = "something"
   done = true
}

func bar(){
   go foo()
   for !done {...}
   y := x
}
```

Using goroutines

- For things which may block or wait
  - EG, on input/output, waiting for stuff to happen, etc
  - Just create as many as you want!
    - Its cheap so why not: let the scheduler do useful work when another one is waiting
- For performance tasks
  - Only create as many as there are CPU cores
    - Otherwise you are wasting resources
  - But it should be more efficient than OpenMP:
    - Thread creation is significantly more overhead than go

Example of Fork/Join Style Parallelism


```
func (v Vector) DoAll(u Vector) {
    numCPU := runtime.NumCPU()   // # of CPUs
    c := make(chan int, numCPU) // buffer the signal for done
    for i := 0; i < numCPU; i++ {
        go v.DoSome(i*len(v)/numCPU, (i+1)*len(v)/numCPU, u, c)
    }
    // Drain the channel.
    for i := 0; i < numCPU; i++ {
        <-c    // wait for one task to complete
    }
}
```

Select

- Select allows you to wait on multiple channels
-
```
select {
    case c <- x:
            x, y = y, x+y
    case d := <- e:
          fmt.Println("Got %v", d)
    default:
          time.Sleep(50 * time.Millisecond)
}
```
- If can write or read to a channel, do so and ***then*** execute the associated case
  - If multiple cases are valid, chose one at random
- If nothing is available, execute default (if any)
  - If no default, just block until you can write or read
  - Default enables non-blocking read & write

Lots of other features for code correctness

- Compiler is, umm, persnickety
  - It is an error to declare but not use a variable or include but not use a package
- Designed to turn comments into documents
  - Including examples
- Libraries for building example and test routines
- Built in package management
- “Single workspace” notion
  - Use common modules to prevent code drift

But one yuge problem...

- Go's exception handling mechanism is dreadful!
  - Instead, the model is C-style: check every function to make sure it returned properly
- Its a little better than C because of multiple return values, but...
  - You will write this EVERYWHERE in your Go code if its good:
    - `res, err := f(...)`
      `if err != nil {...} // Usually just return the err up!`
  - And if its bad...
    - `res, _ := f(...) // Forget it Jake, its ~~Chinatown~~ go error handling`

There is panic, but...

- You can do a **`panic`** and **`recover`**, but...
  - You get to send a *string* with panic
  - **`recover`** is then in a defer block
    - ```
      defer func() {
              if r := recover(); r != nil {
                  ...
              }
      ```
- Which means this results in programming even worse than Python's "catch everything" motif
  - Because you ***only*** have "catch everything"
- Java style is ***much*** better:
  - Exceptions are typed
  - You must either declare or catch all exceptions within a function
- Hopefully 2.0 fixes this...
      ```

Some Real-World Go-Code: String Manipulation


```
// Input format is pairs separated by commas, e.g.
// 123456~foo,123459~bar
func parseRelays(relays string)( r [] RelayInfo) {
        r = make([] RelayInfo,0)
        data := strings.Split(relays, ",")
        for _, i := range(data){ // range builtin can return key/value
                var relay RelayInfo
                splits := strings.Split(i, "~")
                relay.Fingerprint = splits[0]
                if len(splits) > 1 {
                  relay.Name = strings.Split(i,"~")[1]
                } else {relay.Name = "none"}
                r = append(r, relay)
        }
        return r
}
```

Some Real-World Go: 3 separate channels in a receiver loop


```
// Multiple channels:
// Channels: collectIdle/collectAck unbuffered,
// msg/fetchAck unbuffered, packetChannel buffered
func receiver(){
 …
  for true {
    select{
    case idle = <- collectIdle:
      collectAck <- true
    case packet := <- packetChannel:
      if !active && idle{
          idlePackets = append(idlePackets, packet)
      } else {
          packets = append(packets, packet) …
          if (done && ip.SrcIP.String() == idleIP && tcp.RST) {
               …
               analyzePackets(packets, sent, recv)
               fetchAck <- true
               packets = make([]gopacket.Packet, 0, 1024)
      }
    case msg := <- fetchChannel:
      if msg.clear{
           …
           fetchAck <- true
      } else {…
}
```

Switch Topics: Premature Optimization

- There are a lot of opportunities for code optimization...
  - Especially when you are aware of the hardware
- We do this for Project 4:
  - Parallelization with both OpenMP and x86 parallelization ISA
- But you should almost ***never*** spend much time optimizing
  - Amdahl’s Law
  - The Rebugging Problem
  - Algorithms > Microoptimizations

Amdahl's Law and Programmer Effort

- Remember the blindingly obvious: the time the task takes to complete is $T_{programming} + T_{execution}$
  - Unless you are going to run your program a lot
  - Your "runtime" will be limited by your own time
- It makes negative sense to optimize your execution time unless your execution time exceeds your programming time
  - Since often you end up spending vastly more programming time than you'd save
- In the real world, select for programmer productivity first
  - Only if performance really ***really*** matters should you do anything else
  - And even then, make sure you can't just throw $ at the problem:
    A $10k server is a couple weeks of programmer time

The Rebugging Problem

- “The art of programming consists of two tasks, debugging and rebugging” -Me
- If you are optimizing a program…
  - You are ***going*** to be adding new bugs rather than removing old ones
- If your program is working, why do you want to break it?
  - Because odds are, refactoring code to improve performance is going to start with breaking things

Algorithms are far more important than microoptimizations

- Knowing how the cache is laid out is cool and useful
  - But unless you are squeezing out the last 1%, you would be better served focusing on the algorithms
- Project 4 is a great example:
  - 3x performance gain through parallelization with OpenMP…
  - 100x performance gain by going with a better algorithm to do the same task
- I don't care how fast your bubble-sort implementation is…
  - Its still a cruddy $O(n^2)$ implementation