Language-level Concurrency Support: Go

Chris Rossbach
cs378h
Outline for Today

• Questions?

• Administrivia
  • Lab 3 looms large: Go go go!
  • K-Means discussion → office hours

• Agenda
  • Message Passing background
  • Concurrency in Go
  • Thoughts and guidance on Lab 3

• Acknowledgements: Rob Pike’s 2012 Go presentation is excellent, and I borrowed from it: https://talks.golang.org/2012/concurrency.slide
Faux Quiz questions

• How are promises and futures different or the same as goroutines
• What is the difference between a goroutine and a thread?
• What is the difference between a channel and a lock?
• How is a channel different from a concurrent FIFO?
• What is the CSP model?
• What are the tradeoffs between explicit vs implicit naming in message passing?
• What are the tradeoffs between blocking vs. non-blocking send/receive in a shared memory environment? In a distributed one?
Event-based Programming: Motivation

• Threads have a *lot* of down-sides:
  • Tuning parallelism for different environments
  • Load balancing/assignment brittle
  • Shared state requires locks →
    • Priority inversion
    • Deadlock
    • Incorrect synchronization
  • ...

• Events: *restructure programming model to have no threads!*
Message Passing: Motivation

• Threads have a *lot* of down-sides:
  • Tuning parallelism for different environments
  • Load balancing/assignment brittle
  • Shared state requires locks →
  • Priority inversion
  • Deadlock
  • Incorrect synchronization
  • …

• Message passing:
  • *Threads aren’t the problem, shared memory is*
  • *restructure programming model to avoid communication through shared memory (and therefore locks)*
Message Passing

• Threads/Processes send/receive messages

• Three design dimensions
  • Naming/Addressing: how do processes refer to each other?
  • Synchronization: how to wait for messages (block/poll/notify)?
  • Buffering/Capacity: can messages wait in some intermediate structure?
Naming: Explicit vs Implicit
Also: Direct vs Indirect

• Explicit Naming
  • Each process must explicitly name the other party
  • Primitives:
    • send(receiver, message)
    • receive(sender, message)

• Implicit Naming
  • Messages sent/received to/from mailboxes
  • Mailboxes may be named/shared
  • Primitives:
    • send(mailbox, message)
    • receive(mailbox, message)
Synchronization

• Synchronous vs. Asynchronous
  • Blocking send: sender blocks until received
  • Nonblocking send: send resumes before message received
  • Blocking receive: receiver blocks until message available
  • Non-blocking receive: receiver gets a message or null

• If both send and receive block
  • “Rendezvouz”
  • Operation acts as an ordering primitive
  • Sender knows receiver succeeded
  • Receiver knows sender succeeded
  • Particularly appealing in distributed environment

Blocking:
+ simple
+ avoids wasteful spinning
- inflexible
- Can hide concurrency

Non-blocking:
+ maximal flexibility
- error handling/detection tricky
- interleaving useful work non-trivial
Communicating Sequential Processes
Hoare 1978

CSP: language for multi-processor machines
• Non-buffered **message passing**
  • No shared memory
  • **Send/recv** are blocking
• **Explicit naming** of src/dest processes
  • Also called direct naming
  • Receiver *specifies source* process
  • Alternatives: *indirect*
    • Port, mailbox, queue, socket
• **Guarded** commands to let processes wait

← Transputer!
An important problem in the CSP model:

- Processes need to receive messages from different senders
- Only primitive: blocking `receive(<name>, message)`

```c
recv_multi(Q) {
    receive(Q, message)
    receive(R, message)
    receive(S, message)
}
```

Is there a problem with this?
Blocking with Indirect Naming

- Processes need to receive messages from different senders
- *blocking receive* with *indirect naming*
  - Process waits on port, gets first message first message arriving at that port

```
receive(port, message)
```

- OK to block (good)
- Requires indirection (less good)
Non-blocking with Direct Naming

• Processes need to receive messages from different senders

• **Non-blocking receive** with **direct naming**
  • Requires receiver to poll senders

```java
while(…) {
    try_receive(Q, message)
    try_receive(R, message)
    try_receive(S, message)
}
```

Polling (bad)   
No indirection (good)
Blocking and Direct Naming

• How to achieve *it*?

• *CSP provides abstractions/primitives for it*
Alternative / Guarded Commands

Guarded command is *delayed* until either

- *guard succeeds* → cmd executes or
- *guard fails* → command aborts

Alternative command:

- list of one or more guarded commands
- separated by ""||""
- surrounded by square brackets

```
[ x ≥ y -> max:= x || y ≥ x -> max:= y ]
```

Examples

- `n < 10 -> A!index(n); n := n + 1;
  n < 10; A?index(n) = next = MyArray(n);`

Enable *choice* preserving concurrency

- *Hugely influential*
- goroutines, channels, select, defer:
  - *Trying to achieve the same thing*
Go Concurrency

• CSP: the root of many languages
  • Occam, Erlang, Newsqueak, Concurrent ML, Alef, Limbo

• Go is a Newsqueak-Alef-Limbo derivative
  • Distinguished by *first class channel support*
  • Program: *goroutines* communicating through *channels*
  • Guarded and alternative-like constructs in *select* and *defer*
A boring function

```go
func boring(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}

func main() {
    boring("boring!")
}
```

boring! 0
boring! 1
boring! 2
boring! 3
boring! 4
boring! 5
Ignoring a boring function

• Go statement runs the function
• Doesn’t make the caller wait
• Launches a goroutine
• Analogous to & on shell command

• Keep main() around a while
• See goroutine actually running

```go
package main

import (  "fmt"  "math/rand"  "time"
)

func main() {  go boring("boring!")  

fmt.Println("I'm listening.")  time.Sleep(2 * time.Second)  fmt.Println("You're boring; I'm leaving.")
}
```
Goroutines

• Independently executing function launched by go statement
• Has own call stack
• Cheap: Ok to have 1000s...100,000s of them
• Not a thread
  • One thread may have 1000s of go routines!
• Multiplexed onto threads as needed to ensure forward progress
  • Deadlock detection built in
Channels

- Connect goroutines allowing them to communicate
- When main executes <-c, it blocks
- When boring executes c <- value it blocks
- Channels communicate *and synchronize*

```go
func main() {
    c := make(chan string)
    go boring("boring!", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c) // Receive expression is just a value.
    }
    fmt.Println("You're boring; I'm leaving.")
}

func boring(msg string, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprintf("%s %d", msg, i) // Expression to be sent can be any sort.
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

You say: "boring! 0"
You say: "boring! 1"
You say: "boring! 2"
You say: "boring! 3"
You say: "boring! 4"
You're boring; I'm leaving.

Program exited.
Select: Handling Multiple Channels

• All channels are evaluated
• Select blocks until one communication can proceed
  • Cf. Linux select system call, Windows WaitForMultipleObjectsEx
  • Cf. Alternatives and guards in CPS
• If multiple can proceed select chooses randomly
• Default clause executes immediately if no ready channel

```go
select {
  case v1 := <-c1:
    fmt.Printf("received %v from c1\n", v1)
  case v2 := <-c2:
    fmt.Printf("received %v from c2\n", v1)
  case c3 <- 23:
    fmt.Printf("sent %v to c3\n", 23)
  default:
    fmt.Printf("no one was ready to communicate\n")
}
```
Google Search

• Workload:
• Accept query
• Return page of results (with ugh, ads)
• Get search results by sending query to
  • Web Search
  • Image Search
  • YouTube
  • Maps
  • News, etc
• How to implement this?
Search 1.0

- Google function takes query and returns a slice of results (strings)
- Invokes Web, Image, Video search serially

```go
func Google(query string) (results []Result) {
    results = append(results, Web(query))
    results = append(results, Image(query))
    results = append(results, Video(query))
    return
}
```
Search 2.0

• Run Web, Image, Video searches concurrently, wait for results
• No locks, conditions, callbacks

```go
func Google(query string) (results []Result) {
    c := make(chan Result)
    go func() { c <- Web(query) } ()
    go func() { c <- Image(query) } ()
    go func() { c <- Video(query) } ()

    for i := 0; i < 3; i++ {
        result := <-c
        results = append(results, result)
    }
    return
}
```
Search 2.1

• Don’t wait for slow servers: No locks, conditions, callbacks!

```go
    c := make(chan Result)
    go func() { c <- Web(query) } ()
    go func() { c <- Image(query) } ()
    go func() { c <- Video(query) } ()

    timeout := time.After(80 * time.Millisecond)
    for i := 0; i < 3; i++ {
        select {
            case result := <-c:
                results = append(results, result)
            case <-timeout:
                fmt.Println("timed out")
                return
        }
    }
    return
```
Search 3.0

• Reduce tail latency with replication. No locks, conditions, callbacks!
Other tools in Go

- Goroutines and channels are the main primitives
- Sometimes you just need a reference counter or lock
  - “sync” and “sync/atomic” packages
  - Mutex, condition, atomic operations
- Sometimes you need to wait for a go routine to finish
  - Didn’t happen in any of the examples in the slides
  - WaitGroups are key
func testQ() {
    var wg sync.WaitGroup
    ch := make(chan int)
    for i := 0; i < 4; i++ {
        go func(id int) {
            aval, amore := <- ch
            if amore {
                fmt.Printf("reader #%d got %d value\n", id, aval)
            } else {
                fmt.Printf("channel reader #%d terminated with nothing.\n", id)
            }
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
}
Go: magic or threadpools and concurrent Qs?

• We’ve seen several abstractions for
  • Control flow/exeuction
  • Communication

• Lots of discussion of pros and cons

• Ultimately still CPUs + instructions

• Go: just sweeping issues under the language interface?
  • Why is it OK to have 100,000s of goroutines?
  • Why isn’t composition an issue?
Go implementation details

- \( M = \text{“machine”} \rightarrow \text{OS thread} \)
- \( P = \text{(processing) context} \)
- \( G = \text{goroutines} \)
- Each ‘\( M \)’ has a queue of goroutines
- Goroutine scheduling is cooperative
  - Switch out on complete or block
  - Very light weight (fibers!)
  - Scheduler does work
    - stealing

```
struct Sched {
  Lock;       // global sched lock.
  // must be held to edit G or M queues
  G *gfree;   // available g’s (status == Gdead)
  G *ghead;   // g’s waiting to run queue
  G *gtail;   // tail of g’s waiting to run queue
  int32 gwait; // number of g’s waiting to run
  int32 gcount; // number of g’s that are alive
  int32 grunning; // number of g’s running on cpu
    // or in syscall
  M *mhead;   // m’s waiting for work
  int32 mwait; // number of m’s waiting for work
  int32 mcount; // number of m’s that have been created
  ...
};
```
func testQ(consumers int) {
    startTimes["testQ"] = time.Now()
    var wg sync.WaitGroup
    wg.Add(consumers)
    ch := make(chan int)
    for i:=0; i<consumers; i++ {
        go func(id int) {
            aval, amore := <- ch
            if amore {
                info("reader #%d got %d value\n", id, aval)
            } else {
                info("channel reader #%d terminated with nothing.\n", id)
            }
            wg.Done()
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
    stopTimes["testQ"] = time.Now()
}
Channel implementation

- You can just read it: `https://golang.org/src/runtime/chan.go`
- Some highlights

Transputers did this in hardware in the 90s btw.
Channel implementation

• You can just read it:
  • https://golang.org/src/runtime/chan.go

• Some highlights:
  • Race detection built in
  • Fast path just write to receiver stack
  • Often has no capacity → scheduler hint!
  • Buffered channel implementation fairly standard
Go: Sliced Bread 2.0?

- Lacks compile-time generics
  - Results in code duplication
  - Metaprogramming cannot be statically checked
  - Standard library cannot offer generic algorithms
- Lack of language extensibility makes certain tasks more verbose
  - Lacks operator overloading (Java)
- Pauses and overhead of garbage collection
  - Limit Go’s use in systems programming compared to languages with manual memory management
- Right tradeoffs? None of these problems have to do with concurrency!
Now. Let’s discuss Lab 3
Binary Search Trees

- Each node has a value
- Left nodes have smaller values
- Right nodes have greater values

- Want to detect duplicate trees
  - Insertion order affects layout
- Linearize trees for comparison
  - Makes comparison expensive
Hashing BSTs

func initialHash() uint64 {
    return 1
}

func hash(uint64 hash, uint64 val) {
    val2 = val + 2
    prime = 4222234741
    return (hash*val2+val2)%prime
}

• Initialize hash
• Traverse tree in-order
• Incorporate values into hash

• Hash function doesn’t have to be very complex
• Just make sure it handles zeros and similar numbers nicely
Processing pipeline

- Read in trees from file
  - Array / slice of BSTs
- Hash trees + insert hashes
  - Map from hash to tree indexes
- Compare trees
  - Equivalence matrix
    - num trees x num trees
Parallelizing the pipeline

Step 2
- Implement just hashing first
- Goroutines
  - 1 per tree
  - Dedicated inserter goroutine(s)
    - Communicate via channel
- Thread pool
  - hash-workers threads
  - Acquire lock(s) to insert
- Multiple data-workers optional

Step 3
- Goroutines
  - 1 per comparison
- Thread pool
  - comp-workers threads
  - Send work via channel
    - (Optional) custom implementation
    - Queue, mutex, and conditions
- Store results directly in matrix
Go: command-line flags

```go
import "flag"

func main() {
    intptr = flag.Int("num", 0, "number argument")
    flag.Parse()
    num := *flagPtr
}

./my_program -num=1
```
Go: file parsing

```go
import ("io/ioutil" "strconv" "strings")
func main() {
    fileData, err := ioutil.ReadFile(fileName)
    fileData = fileData[:len(fileData)-1]   // remove EOF
    fileLines := strings.Split(string(fileData), "\n")
    for _, line := range fileLines {
        // parse line with strings.Split and strconv.Atoi()
    }
}
```
import "time"

func main() {
    start := time.Now()

    // do some work

    timeTakenStr := time.Since(start)

    fmt.Printf("Doing work took %s\n", timeTakenStr)
}

Go: timing
Go: functions and return values

```go
func notMain() (int, bool) {  // multiple return values
    return (3, false)
}

func main() {
    i, b := notMain()
    j, _ := notMain()   // throw away value
}
```
Go: synchronization

import "sync"   // contains WaitGroups
func main() {
    var *mutex = &sync.Mutex{}   // pointer to mutex
    var *cond = &sync.NewCond(mutex)   // mutex condition
    mutex.Lock()
    cond.Wait()   // releases lock on mutex
    cond.Signal()   // wakes threads waiting on cond
    mutex.Unlock()
}
func main() {
    mySlice := make([]int, 2)
    mySlice[1] = 5   // can use like an array
    mySlice = append(mySlice, 10)   // can use like a list
    l := len(mySlice)
    subSlice := mySlice[0:1]   // can slice like in Python
    fromStartToTwo := mySlice[:2]
    fromOneToEnd := mySlice[1:]
}
Go: maps

```go
func main() {
    mapIntBool := make(map[int] bool) // map from ints to bools
    mapIntBool[5] = true
    for key, value := range mapIntBool {
        // use key or value
    }
}
// map value can be a slice
```
type myStruct struct {
    mySlice []int
    myChan chan int
    mySliceOfSlice [][]bool
    myPtr *myStruct
}

var ms myStruct  // declare variable without initialization
// use dot operator for structs, pointers, and pointers to structs
ms.myPtr.mySlice[2]