Potpourri

Doug Woos

Logistics notes

Piazza!!!

https://piazza.com/washington/spring2017/cse452

In-class questions

Outline

- More Go
- Remote procedure calls
- MapReduce discussion

More Go

Hopefully you got the basics from section

Today:

- Doug’s go tips
- Synchronization
- Remote procedure calls

Goroutines

Lightweight (“green”) threads

Multiplexed onto $GOMAXPROCS OS threads

If they block, make an OS thread

Convenient syntax—if you realize you want to do something async, just add “go”

If/else

This is wrong:

```
if x > 0 {
  // something
}
else {
  // something else
}
```

This is right:

```
if x > 0 {
  // something
} else {
  // something else
}
```

Anonymous functions

Handy when using go-routines

```
go func() {
  // do some work
}()
```

But: careful with arguments

What does this do?

```
for val := range values {
  go func() {
    fmt.Println(val)
  }()
}
```

Anonymous functions

Handy when using go-routines

```
go func() {
  // do some work
}()
```

But: careful with arguments

What does this do?

```
for val := range values {
  go func(val) {
    fmt.Println(val)
  }(val)
}
```

Communicating Sequential Processes

Hoare's model for concurrency

Locks (monitors): multiple threads access data, making sure to acquire lock

CSP: one thread accesses data, other threads communicate via channels

Use either, but not both for same data

For this lab, just use channels

Subsequent labs built around locks

Locking

Mutexes in “sync” library—sync.mutex

```
import “sync”

type Data struct {
  mu sync.mutex
}
```

```
func (wk *Worker) accessData(…) {
  wk.mu.Lock()
  defer wk.mu.Unlock()
}
```

Advice: develop and follow a coherent system

Lock at top level, require subroutines to be called with lock held (and add comments to that effect)

Remote procedure calls

Request from a client to execute a function on a server

Basic communication technique

Today: Basic concepts, usage in lab 1

Next time: RPC semantics in detail

Remote procedure calls

Differences between RPC and local call

- Need to bind to server (like linking)

- Performance

- Failures—msg drop, client crash, server crash, slowness

RPC implementation

```
ok := call(address, "Worker.DoJob",
           args, &reply)
```

RPC library

Serialize args
Open connection
Write data

Read data
Deserialize reply

OS

TCP/IP write

TCP/IP read

Transport

```
func (wk *Worker)
     DoJob(args *DJArgs,
           reply *DJReply)
```

RPC library

Serialize reply
Write data

Read data
Deserialize args

OS

TCP/IP write

TCP/IP read

Transport

CSE 461

RPC in Labs

Go “rpc” library

We wrap it in some convenience functions

You won’t have to manually register RPCs

Important later: interface{} works fine

Capitalization weirdly important

- Capitalized fields on structs sent
- Capitalized methods registered as RPCs

Go RPCs: Server-side

RPCs have two args and return error code (or nil)

```
func Funcname(arg *FuncArgs, reply *FuncReply) error
```

(You can’t get the error, so just return nil)

Go RPCs: Client-side

call function

```
ok := call(address, “Type.Method”, args, &reply)
```

Returns a bool

If ok is false, did the call happen?

- For this lab, assume no

- In future labs, ???

RPCs in Lab 1

Worker and master communicate with each other

Worker->master: registration

```
func (mr *MapReduce) Register(args *RegisterArgs,
                              res *RegisterReply) error
```

Master->worker: DoJob(map or reduce), Shutdown

```
func (wk *Worker) DoJob(arg *DoJobArgs,
                        res *DoJobReply) error

func (wk *Worker) Shutdown(args *ShutdownArgs,
                           res *ShutdownReply) error
```

RPCs and Concurrency

Blocking on the client

- MapReduce master has multiple outstanding jobs

Need thread per worker or thread per RPC

Keep track of which jobs have been done

Only start Reduce tasks once Map tasks done

For part 3: put tasks back on queue if they fail

RPCs and Concurrency

Concurrent on the server

Not an issue in lab 1

In subsequent labs, need to lock

MapReduce Discussion

What's the deal with master failure?

Why is atomic rename important?

Why not store intermediate results in RAM?

- Apache Spark

Aren't some Reduce jobs much larger?

What about infinite loops?

Why does novelty matter?

Since we have some time

I claimed that a Two Generals protocol is impossible

Why?