Lecture 13 - Go Concurrency / Start of Smart Contracts

Concurrency in Go

Go routines

A Go Routine:

```
        go func( a int ) {
                fmt.Printf ( "a=%d\n", a )
        }( 12 )
```

10 Go Routines:

```
package main

import "fmt"

func main() {
        for i := 0; i < 10; i++ {
                go func(a int) {
                        fmt.Printf("a=%d\n", a)
                }(12 + i)
        }
}
```

With NO output - Why?

```
package main

import (
        "fmt"
        "sync"
)

func main() {
    var wg sync.WaitGroup
        for i := 0; i < 10; i++ {
```

```
                wg.Add(1)
                go func(a int) {
                        defer wg.Done()
                        fmt.Printf("a=%d\n", a)
                }(12 + i)
        }
    wg.Wait()
}
```

And OUTPUT!!!!

```
$ go run go-r2.go
a=17
a=19
a=12
a=21
a=13
a=16
a=14
a=20
a=15
a=18
```

locks

From: https://nathanleclaire.com/blog/2014/02/15/how-to-wait-for-all-goroutines-to-finish-executing-before-continuing/

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    messages := make(chan int)
    var wg sync.WaitGroup

    // you can also add these one at
    // a time if you need to

    wg.Add(3)
    go func() {
        defer wg.Done()
```

```
        time.Sleep(time.Second * 3)
        messages <- 1
    }()
    go func() {
        defer wg.Done()
        time.Sleep(time.Second * 2)
        messages <- 2
    }()
    go func() {
        defer wg.Done()
        time.Sleep(time.Second * 1)
        messages <- 3
    }()
    go func() {
        for i := range messages {
            fmt.Println(i)
        }
    }()

    wg.Wait()
}
```

Also “map”’s are not concurrency protected. You have to lock/unlock them yourself.

Problems like this are easy to find. There is a “race detector” built into go and you can run it as a part of your tests.

You should decide if you need to protect a map. Why? When?

```
package main

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

func main() {
        state := make(map[int]int)
        mutex := &sync.Mutex{}

        var nRead uint64
        var nWrite uint64

        const randRange = 15

        for ii := 0; ii < 100; ii++ {
```

```
                go func() {
                        total := 0
                        for {
                                key := rand.Intn(randRange)
                                mutex.Lock()
                                total += state[key]
                                mutex.Unlock()
                                atomic.AddUint64(&nRead, 1)
                                time.Sleep(time.Millisecond)
                        }
                }()
        }
        for jj := 0; jj < 50; jj++ {
                go func() {
                        for {
                                key := rand.Intn(randRange)
                                val := rand.Intn(100)
                                mutex.Lock()
                                state[key] = val
                                mutex.Unlock()
                                atomic.AddUint64(&nWrite, 1)
                                time.Sleep(time.Millisecond)
                        }
                }()
        }

        time.Sleep(time.Second * 1)

        nReadTotal := atomic.LoadUint64(&nRead)
        nWriteTotal := atomic.LoadUint64(&nWrite)

        mutex.Lock()
        fmt.Printf("ReadOps: %d\nWriteOps: %d\nFinal State: %+v\n", nReadTotal, nWriteT
        mutex.Unlock()

}
```

The Output

```
$ go run atomic.go
ReadOps: 81881
WriteOps: 40936
Final State: map[10:70 3:81 9:81 12:55 5:67 1:38 6:89 14:28 0:40 8:13 4:11 13:19 2:40 1
$ go run atomic.go
ReadOps: 82500
WriteOps: 41250
Final State: map[2:34 10:2 4:28 5:80 14:42 0:46 3:55 1:65 12:63 9:10 13:50 7:17 6:19 11
```

channels

```
package main

import (
        "fmt"
        "os"
        "sync"
        "time"
)

func main() {
        msg := make(chan string)
        msg2 := make(chan string)
        var wg sync.WaitGroup
        for i := 0; i < 10; i++ {
                wg.Add(1)
                go func(n int) {
                        for {
                                time.Sleep(time.Millisecond * 50)
                                msg <- fmt.Sprintf("ping:%d", n)
                        }
                }(i)
        }
        for i := 0; i < 10; i++ {
                wg.Add(1)
                go func(n int) {
                        for {
                                time.Sleep(time.Millisecond * 55)
                                msg2 <- fmt.Sprintf("PONG:%d", n)
                        }
                }(i)
        }
        nMsg := 0
        for {
                select {
                case out := <-msg:
                        nMsg++
                        fmt.Printf("%s\n", out)
                case out := <-msg2:
                        nMsg++
                        fmt.Printf("%s\n", out)
                }
                if nMsg > 100 {
                        os.Exit(0)
                }
        }
```

```
        wg.Wait()
}
```

Output:

```
$ go run chan.go
ping:1
ping:6
ping:0
ping:5
ping:9
ping:7
ping:8
ping:2
ping:3
ping:4
PONG:3
PONG:5
PONG:4
PONG:8
PONG:7
PONG:1
PONG:2
PONG:9
PONG:6
PONG:0
ping:1
...
```

Smart Contracts - Standard Contracts (ERC-20)

Standard ERC-20 Contract

SimpleToken

Method Name	Const	$	Params
Approval	event		(address owner, address spender, uint256 value)
INITIAL_SUPPLY	const		() returns (uint256)
Transfer	event		(address from, address to, uint256 value)
allowance	const		(address _owner, address _spender) returns (uint256)

Method Name	Const	$	Params
approve	Tx		(address _spender, uint256 _value) returns (bool)
balanceOf	const		(address _owner) returns (uint256)
decimals	const		() returns (uint8)
decreaseApproval	Tx		(address spender, uint256 _subtractedValue) returns (bool)
increaseApproval	Tx		(address _spender, uint256 _addedValue) returns (bool)
name	const		() returns (string)
symbol	const		() returns (string)
totalSupply	const		() returns (uint256)
transfer	Tx		(address _to, uint256 _value) returns (bool)
transferFrom	Tx		(address from. address _to, uint256 _value) returns (bool)
constructor	()		

SimpleToken Ours derived from StandardToken

```
pragma solidity ^0.4.24;

import "openzeppelin-solidity/contracts/token/ERC20/StandardToken.sol";

/**
 * @title SimpleToken
 * @dev Very simple ERC20 Token example, where all tokens are pre-assigned to the creat
 * Note they can later distribute these tokens as they wish using `transfer` and other
 * `StandardToken` functions.
 */
contract SimpleToken is StandardToken {

        string public constant name = "SimpleToken"; // solium-disable-line uppercase
        string public constant symbol = "SIM"; // solium-disable-line uppercase
        uint8 public constant decimals = 0; // solium-disable-line uppercase

        uint256 public constant INITIAL_SUPPLY = 10000 * (10 ** uint256(decimals));

        /**
         * @dev Constructor that gives msg.sender all of existing tokens.
         */
        constructor() public {
                totalSupply_ = INITIAL_SUPPLY;
```

```
                balances[msg.sender] = INITIAL_SUPPLY;
                emit Transfer(0x0, msg.sender, INITIAL_SUPPLY);
        }

}
```

StandardToken

```
pragma solidity ^0.4.24;

import "./BasicToken.sol";
import "./ERC20.sol";


/**
 * @title Standard ERC20 token
 *
 * @dev Implementation of the basic standard token.
 * https://github.com/ethereum/EIPs/issues/20
 * Based on code by FirstBlood: https://github.com/Firstbloodio/token/blob/master/smart
 */
contract StandardToken is ERC20, BasicToken {

  mapping (address => mapping (address => uint256)) internal allowed;


  /**
   * @dev Transfer tokens from one address to another
   * @param _from address The address which you want to send tokens from
   * @param _to address The address which you want to transfer to
   * @param _value uint256 the amount of tokens to be transferred
   */
  function transferFrom(
    address _from,
    address _to,
    uint256 _value
  )
    public
    returns (bool)
  {
    require(_to != address(0));
    require(_value <= balances[_from]);
    require(_value <= allowed[_from][msg.sender]);

    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(_value);
    allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
    emit Transfer(_from, _to, _value);
    return true;
  }

  /**
   * @dev Approve the passed address to spend the specified amount of tokens on behalf
   * Beware that changing an allowance with this method brings the risk that someone ma
   * and the new allowance by unfortunate transaction ordering. One possible solution t
   * race condition is to first reduce the spender's allowance to 0 and set the desired
```

```
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     * @param _spender The address which will spend the funds.
     * @param _value The amount of tokens to be spent.
     */
    function approve(address _spender, uint256 _value) public returns (bool) {
      allowed[msg.sender][_spender] = _value;
      emit Approval(msg.sender, _spender, _value);
      return true;
    }

    /**
     * @dev Function to check the amount of tokens that an owner allowed to a spender.
     * @param _owner address The address which owns the funds.
     * @param _spender address The address which will spend the funds.
     * @return A uint256 specifying the amount of tokens still available for the spender.
     */
    function allowance(
      address _owner,
      address _spender
     )
      public
      view
      returns (uint256)
    {
      return allowed[_owner][_spender];
    }

    /**
     * @dev Increase the amount of tokens that an owner allowed to a spender.
     * approve should be called when allowed[_spender] == 0. To increment
     * allowed value is better to use this function to avoid 2 calls (and wait until
     * the first transaction is mined)
     * From MonolithDAO Token.sol
     * @param _spender The address which will spend the funds.
     * @param _addedValue The amount of tokens to increase the allowance by.
     */
    function increaseApproval(
      address _spender,
      uint256 _addedValue
    )
      public
      returns (bool)
    {
      allowed[msg.sender][_spender] = (
        allowed[msg.sender][_spender].add(_addedValue));
      emit Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
      return true;
    }

    /**
     * @dev Decrease the amount of tokens that an owner allowed to a spender.
```

```
     * approve should be called when allowed[_spender] == 0. To decrement
     * allowed value is better to use this function to avoid 2 calls (and wait until
     * the first transaction is mined)
     * From MonolithDAO Token.sol
     * @param _spender The address which will spend the funds.
     * @param _subtractedValue The amount of tokens to decrease the allowance by.
     */
    function decreaseApproval(
      address _spender,
      uint256 _subtractedValue
    )
      public
      returns (bool)
    {
      uint256 oldValue = allowed[msg.sender][_spender];
      if (_subtractedValue > oldValue) {
        allowed[msg.sender][_spender] = 0;
      } else {
        allowed[msg.sender][_spender] = oldValue.sub(_subtractedValue);
      }
      emit Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
      return true;
    }

  }
```

BasicToken

```
pragma solidity ^0.4.24;


import "./ERC20Basic.sol";
import "../../math/SafeMath.sol";


/**
 * @title Basic token
 * @dev Basic version of StandardToken, with no allowances.
 */
contract BasicToken is ERC20Basic {
  using SafeMath for uint256;

  mapping(address => uint256) balances;

  uint256 totalSupply_;
```

```
  /**
  * @dev Total number of tokens in existence
  */
  function totalSupply() public view returns (uint256) {
    return totalSupply_;
  }

  /**
  * @dev Transfer token for a specified address
  * @param _to The address to transfer to.
  * @param _value The amount to be transferred.
  */
  function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender]);

    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    emit Transfer(msg.sender, _to, _value);
    return true;
  }

  /**
  * @dev Gets the balance of the specified address.
  * @param _owner The address to query the the balance of.
  * @return An uint256 representing the amount owned by the passed address.
  */
  function balanceOf(address _owner) public view returns (uint256) {
    return balances[_owner];
  }

}
```

ERC20

```
pragma solidity ^0.4.24;

import "./ERC20Basic.sol";


/**
 * @title ERC20 interface
 * @dev see https://github.com/ethereum/EIPs/issues/20
 */
contract ERC20 is ERC20Basic {
  function allowance(address owner, address spender)
    public view returns (uint256);
```

```
    function transferFrom(address from, address to, uint256 value)
      public returns (bool);

    function approve(address spender, uint256 value) public returns (bool);
    event Approval(
      address indexed owner,
      address indexed spender,
      uint256 value
    );
  }
```