

Concurrency Programming

COS 316 Precept 10

Today's Plan

- Background on concurrency
- Key Golang mechanisms for developing concurrent programs (used in assignment 5)

Background: Overview of Concurrency

Sequential programs:

- Single thread of control
- Subprograms / tasks - don't overlap in time - executed one after another

Concurrent programs

- Multiple threads of control
- Subprograms / tasks - may (conceptually) overlap in time - (appear to be) executed at the same time

- Computer with a single processor can have multiple processes at once
- OS schedules different processes - giving illusion that multiple processes are running simultaneously
- Note - parallel architectures can have N processes running simultaneously on N processors

Background: Operating System (Review)

- Allows many processes to execute concurrently
- Ensures each process' physical address space does not overlap
- Ensures all processes get fair share of processor time and resources
- Processes can run concurrently and (context) switch
- User's perspective: appears that processes run in parallel although they don't

Background: Context Switch

- Control flow changes from one process to another
 - E.g., switching from processA to processB
- Overhead:
 - Before each switch OS has to save the state (context) of currently running process and restore it when next time its execution gets resumed

Background: Threads vs Processes

- Processes
 - Process context switching time is long
(change of *virtual* address space & other resources)
- Threads
 - thread is a “lightweight” process
 - thread shares some of the context with other threads in a process, e.g.
 - Virtual memory
 - File descriptors
- Private context for each thread:
 - Stack
 - Data registers
 - Code (PC)
- Switching between threads is faster because there is less context
 - less data that has to be read/written from/to memory

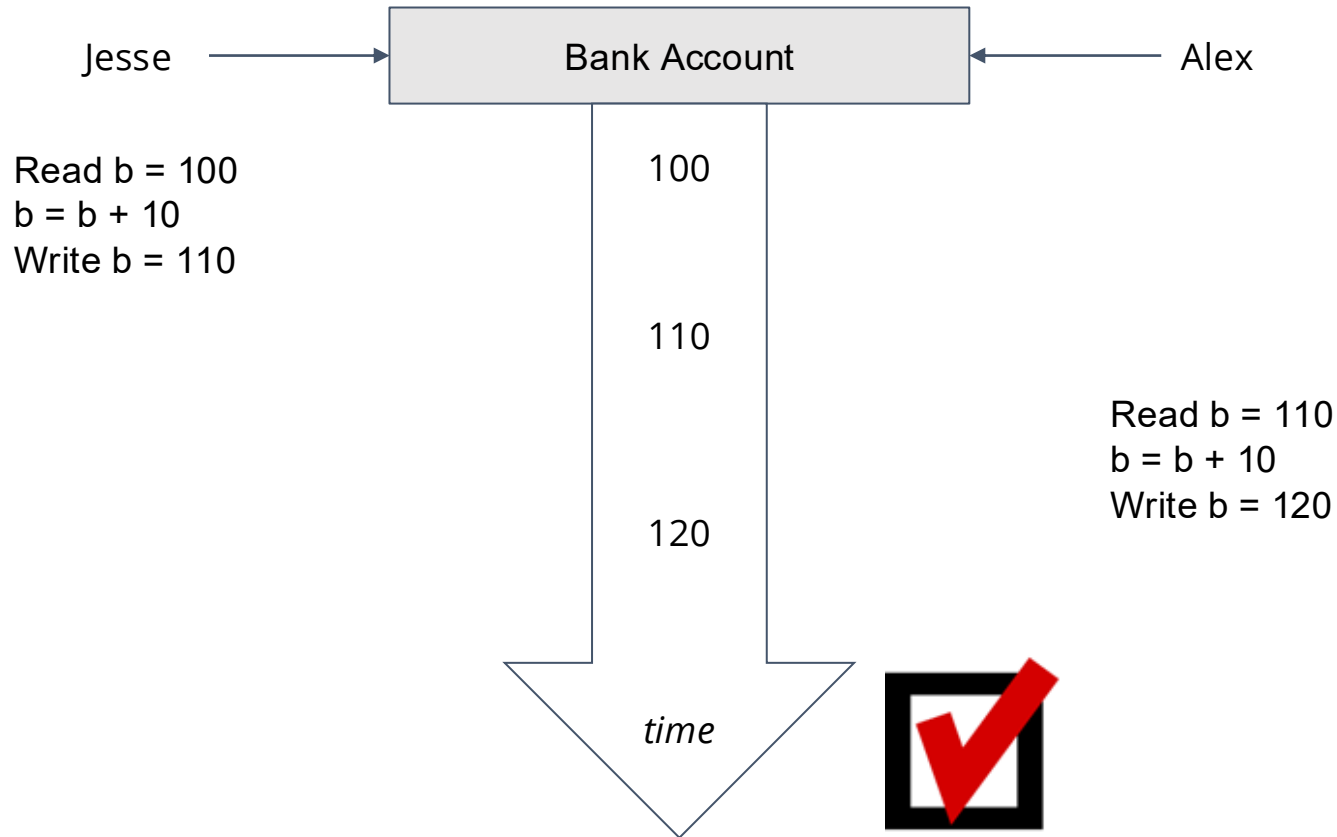
Background: Why Concurrency?

- Performance gain
 - Google search queries
- Application throughput
 - Throughput = amount of work that a computer can do in a given time period
 - When one task is waiting (blocking) for I/O another task can continue its execution
- Model real-world structures
 - Multiple sensors
 - Multiple events
 - Multiple activities

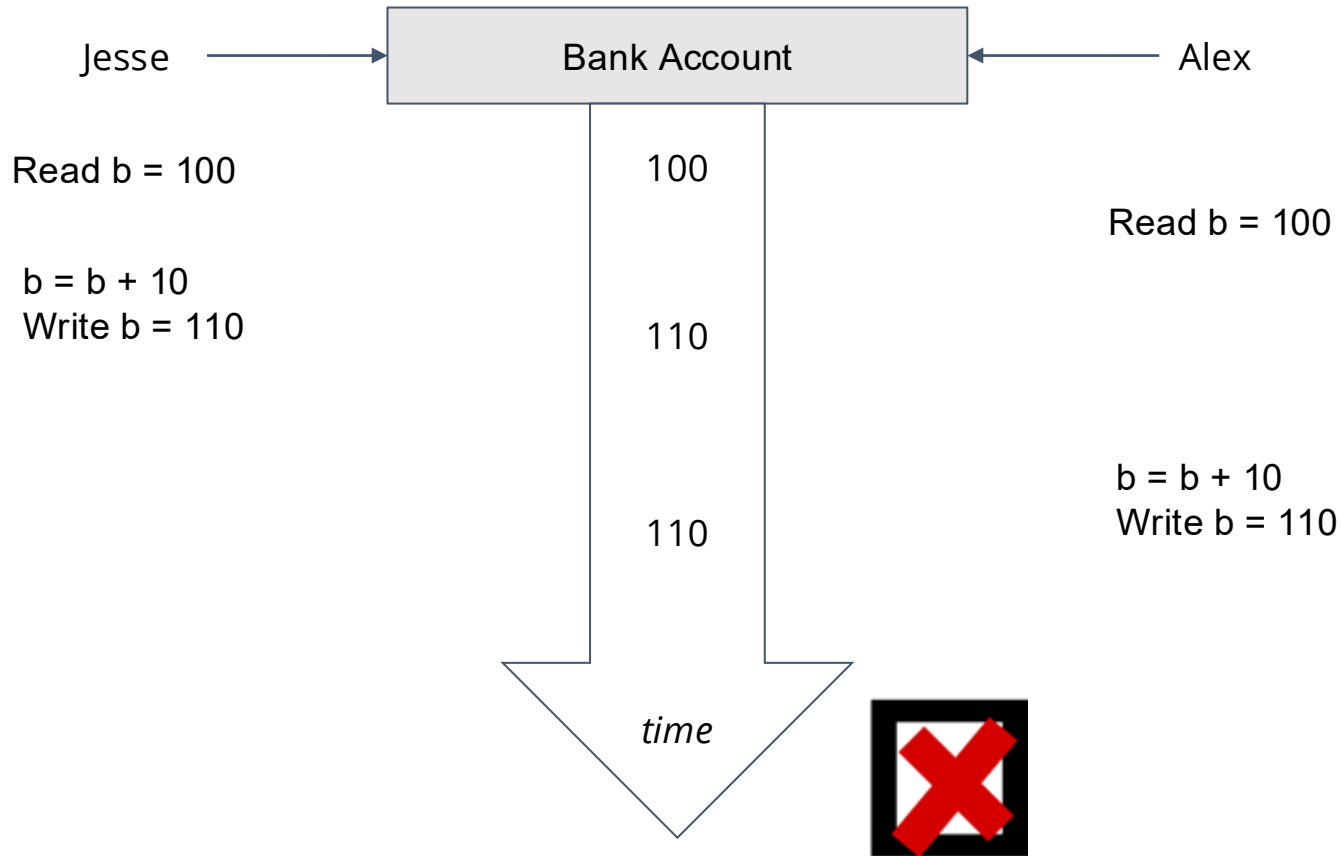
Tradeoffs - Concurrent Programming

- Complex
- Error-prone
- Hard to debug

Example



Example



Go and Concurrency

- Goroutines
- The sync package - <https://golang.org/pkg/sync>
 - sync.Mutex
 - sync.Cond

Goroutines

- A lightweight thread managed by the Go runtime
- Many goroutines execute within a single OS thread
 - One goroutine is created automatically to execute the `main()`
 - Other goroutines are created using the **go** keyword
 - Order of execution depends on the Go scheduler
 - Go takes a process with main thread and schedules / switches goroutines within that thread

- Compare
 - Sequential Program
 - <https://play.golang.org/p/PLeCGtRp2QB>
 - Concurrent program
 - https://play.golang.org/p/sDitCEr_3vX

Goroutines - Exiting

- goroutine exits when code associated with its function returns
- When the main goroutine is complete, **all** other goroutines exit, even if they are not finished
 - goroutines are forced to exit when main goroutine exits
 - goroutine may not complete its execution because main completes early
- Execution order of goroutines is non-deterministic

A simple example to show non-determinism

- https://play.golang.org/p/sDitCEr_3vX
- Switch the order of the calls from

```
go say("world")  
say("hello")
```

```
say("hello")  
go say("world")
```

- What happens?
- How to fix?

Synchronization

- Synchronization is when multiple threads agree on a timing of an event
- Global *events* whose execution is viewed by all threads, simultaneously
- One goroutine does not know the timing of other goroutines
- Synchronization can introduce some global events that every thread sees at the same time

Synchronization and Go

- type Cond
 - Func (*Cond) Signal()
 - func (*Cond) Broadcast()
 - func (*Cond) Wait()

- type Mutex
 - func (m *Mutex) Lock()
 - func (m *Mutex) Unlock()

- Channels
 - See COS 418

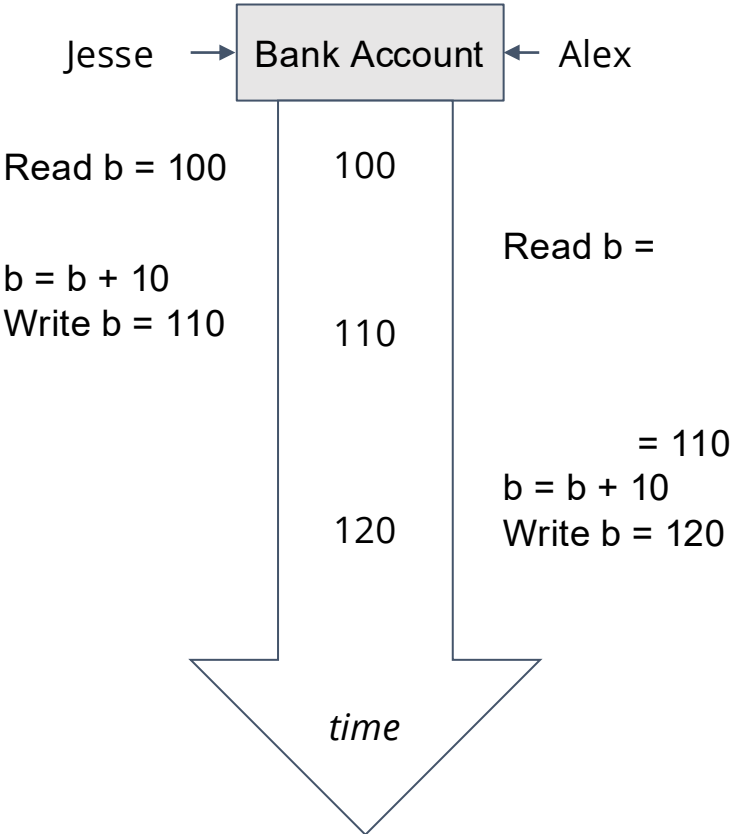
Mutex (Mutual Exclusion)

- Sharing variables between goroutines (concurrently) can cause problems
- Two goroutines writing to the same shared variable can interfere with each other
- Function/goroutine is said to be concurrency-safe if can be executed concurrently with other goroutines without interfering improperly with them
 - e.g., it will not alter variables in other goroutines in some unexpected/unintended/unsafe way

Sync.Mutex

- A mutex ensures *mutual exclusion*
- Uses a binary semaphore
 - If flag is up → shared variable is in use by somebody
- Only one goroutine can write into variable at a time
- Once goroutine is done with using shared variable it has to put the flag down
 - if flag is down → shared variable is available
- If another goroutine see that flag is down it knows it can use the shared variable but first it has to put the flag up

Back to our example



```
func Deposit(amount) {  
    lock balanceLock  
    read balance  
    balance = balance +  
amount  
    write balance  
    unlock balanceLock  
}
```

} CRITICAL SECTION

Sync.Mutex

- **Lock()**
 - Puts the flag up (if none of other goroutines has already put the flag up)
 - If second goroutine also calls **Lock()** it will be blocked, it has to wait until first goroutine releases the lock
 - Note - any number of goroutines (not just two) competing to **Lock()**
 - **Unlock()**
 - Puts the flag down
 - When **Unlock()** is called, a blocked **Lock()** can proceed
 - In general: put **Lock()** at the beginning of the critical section and call **Unlock()** at the end of it; ensures that only one goroutine will be in critical section region
- Create a Mutex
var mut sync.Mutex
 - To lock a critical section
mut.Lock()
 - To unlock a critical section
mut.Unlock()

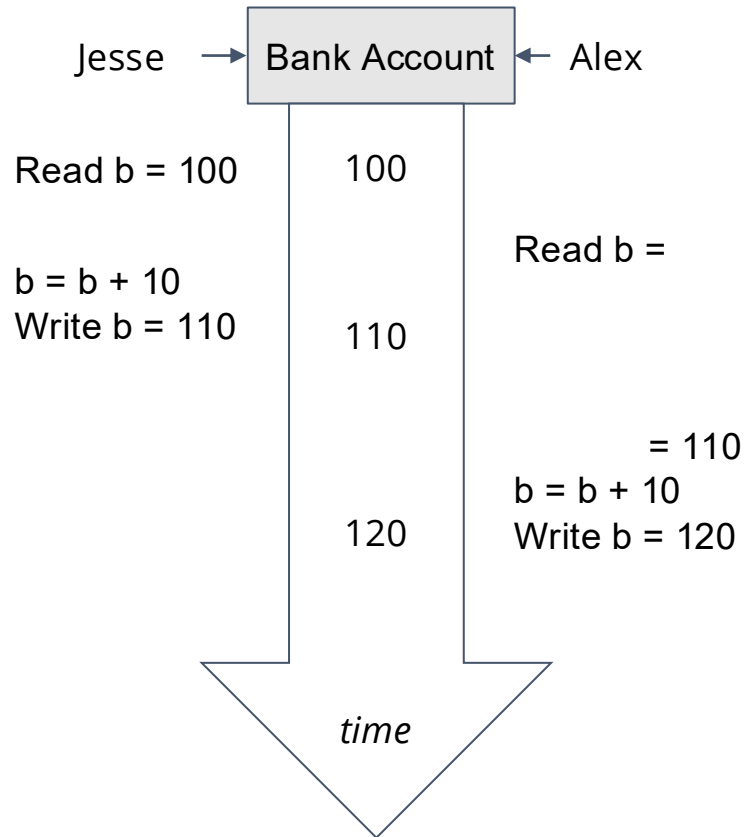
Mutex Exercise

Consider:

```
var i int = 0
var wg sync.WaitGroup
func inc() {
    i = i + 1
    wg.Done()
}
func main() {
    wg.Add(2)
    go inc()
    go inc()
    wg.Wait()
    fmt.Println(i)
}
```

- Run the program
<https://play.golang.org/p/hNevYkKDp30>
- Is it concurrency-safe?
- Use `Lock()` and `Unlock()` to make these programs concurrency-safe

Mutex Exercise - Bank Account



- Make this code concurrency-safe

<https://go.dev/play/p/VboCb85otn0>

Interesting Example

Consider:

```
var mu sync.Mutex
func funcA() {
    mu.Lock()
    funcB()
    mu.Unlock()
}
func funcB() {
    mu.Lock()
    fmt.Println("Hello, World")
    mu.Unlock()
}
func main() {
    funcA()
}
```

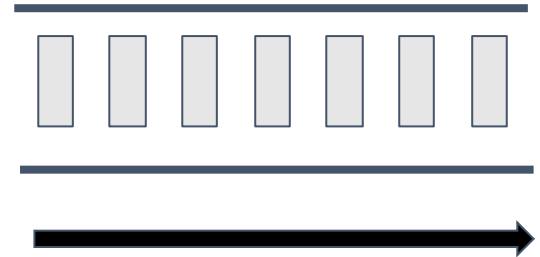
- Run the program

https://play.golang.org/p/c2Qgo-W_4mP

- What happens?

Condition Variables - `sync.Cond`

- [`sync.Cond`](#) type - provides an efficient way to send notifications among goroutines
- `sync.Cond` value holds a [`sync.Locker`](#) field with name `L`
 - field value is of type `*sync.Mutex` or `*sync.RWMutex`
 - E.g.:
 - `cond := sync.NewCond(&sync.Mutex{})`
 - `cond.L.Lock()`
 - `cond.L.Unlock()`
- `sync.Cond` value holds a FIFO queue of waiting goroutines
- commonly used to allow threads to wait on a *condition* to be true: consumers *wait* until a producer *signals* that something happened



Condition Variables - L.Lock(), L.Unlock(), Wait(), Broadcast(), Signal()

- `cond := sync.NewCond(&sync.Mutex{})`
 - `cond.L.Lock()`
 - `cond.Wait()`
 - `cond.Broadcast()`
 - `cond.Signal()`
-
- Call L.Lock() before Wait()
 - Insert calling goroutine in queue and block (wait)
 - Calls L.Unlock()
 - Unblock *all* the goroutines in (and remove them from) the waiting goroutine queue
 - Unblock the head goroutine in (and remove them from) the waiting goroutine queue
 - Blocked routines go back to running state
 - Invokes `cond.L.Lock()` (in the resumed `cond.Wait()` call) to try to acquire and hold the lock `cond.L` again
 - `cond.Wait()` call exits after the `cond.L.Lock()` call returns

A Basic Example:

CONDITION VARIABLE: WAIT AND SIGNAL

```
func (q *Queue) Get() Item {
    q.mu.Lock()
    defer q.mu.Unlock()
    for len(q.items) == 0 {
        q.itemAdded.Wait()
    }
    item := q.items[0]
    q.items = q.items[1:]
    return item
}
```

```
func (q *Queue) Put(item Item) {
    q.mu.Lock()
    defer q.mu.Unlock()
    q.items = append(q.items, item)
    q.itemAdded.Signal()
}
```

Wait atomically unlocks the mutex and suspends the goroutine.
Signal locks the mutex and wakes up the goroutine.

sync.Cond – Always Check the Condition!

- Why is this loop here?
- `cond.Wait()` does not guarantee the condition holds when it returns
- The condition could have been made false again while the goroutine was waiting to run
- Always check the condition, and keep waiting if it does not hold

```
checkCondition := func() bool {  
    // Check the condition  
}  
  
for !checkCondition() {  
    cond.Wait()  
}  
cond.L.Unlock()
```