COS 316 Precept: Concurrency

Precept Objectives

- Learn about basics of concurrent programs and programming
- Understand how to use a few Golang mechanisms for developing concurrent programs

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

- Recall from lecture
  - 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

Operating System - Review

- Allows many processes to execute concurrently
- Ensures each process’s 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

Context Switch - Review

- 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

Scheduling Policies - Discussion

- What are the three policies outlined in lecture?

- What are some other possible policies?

Threads vs Processes

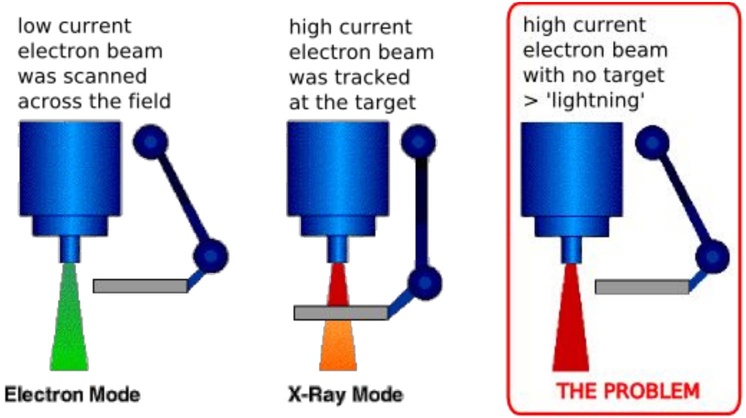
- Processes
  - Process context switching time is long (switching between processes is slow because memory access)
- Threads
  - thread is a “lightweight” process
  - thread shares some of the context with other threads in a process
- Shared process context among process threads:
  - Virtual memory
  - 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

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

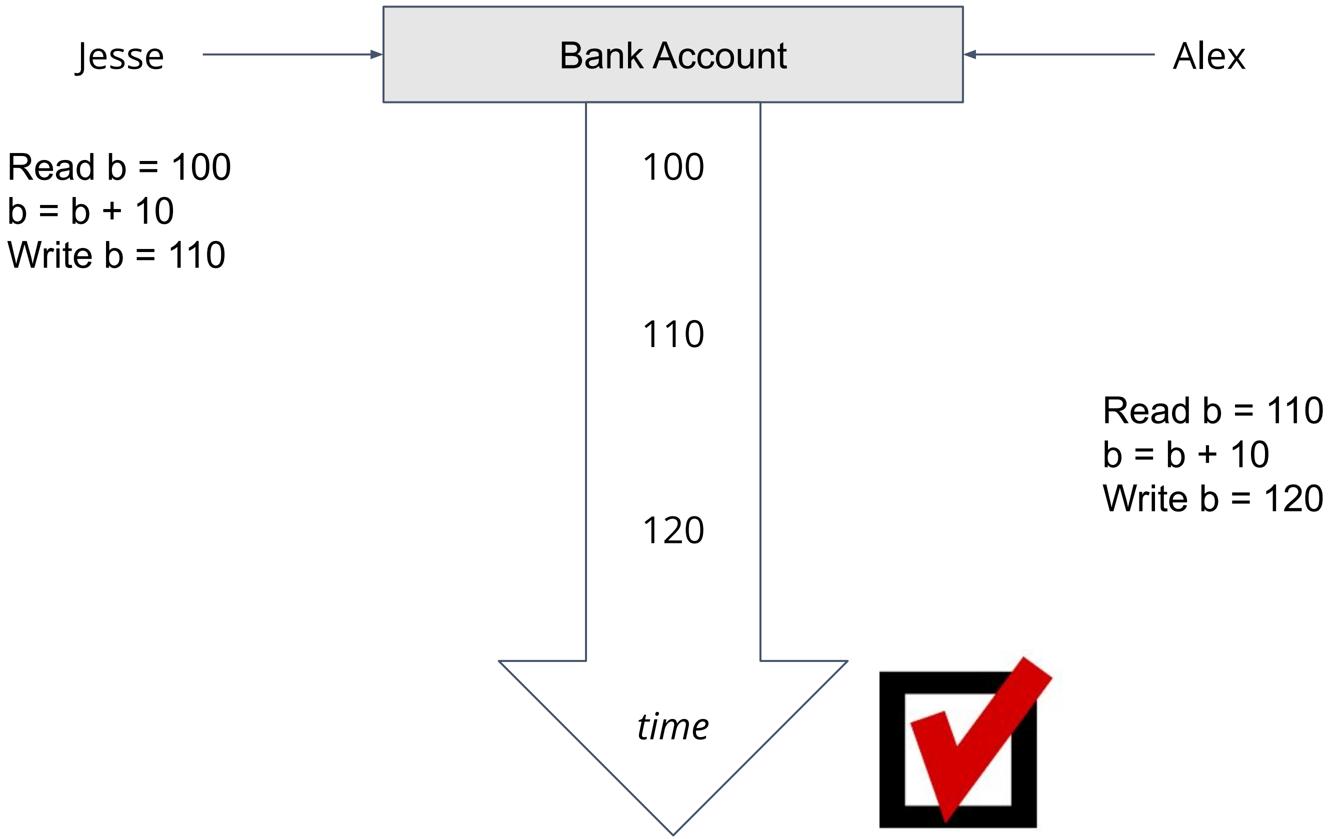
Tradeoffs - Concurrent Programming

- Complex
- Error-prone
- Hard to debug
- Famous bugs related to concurrency:
  - Therac-25
    - Computerized radiation therapy machine
    - Two “safe modes” of operations, but a race condition caused a severely unsafe mode
    - Patients died from overdoses
    - See N.G. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. Computer. July 1993

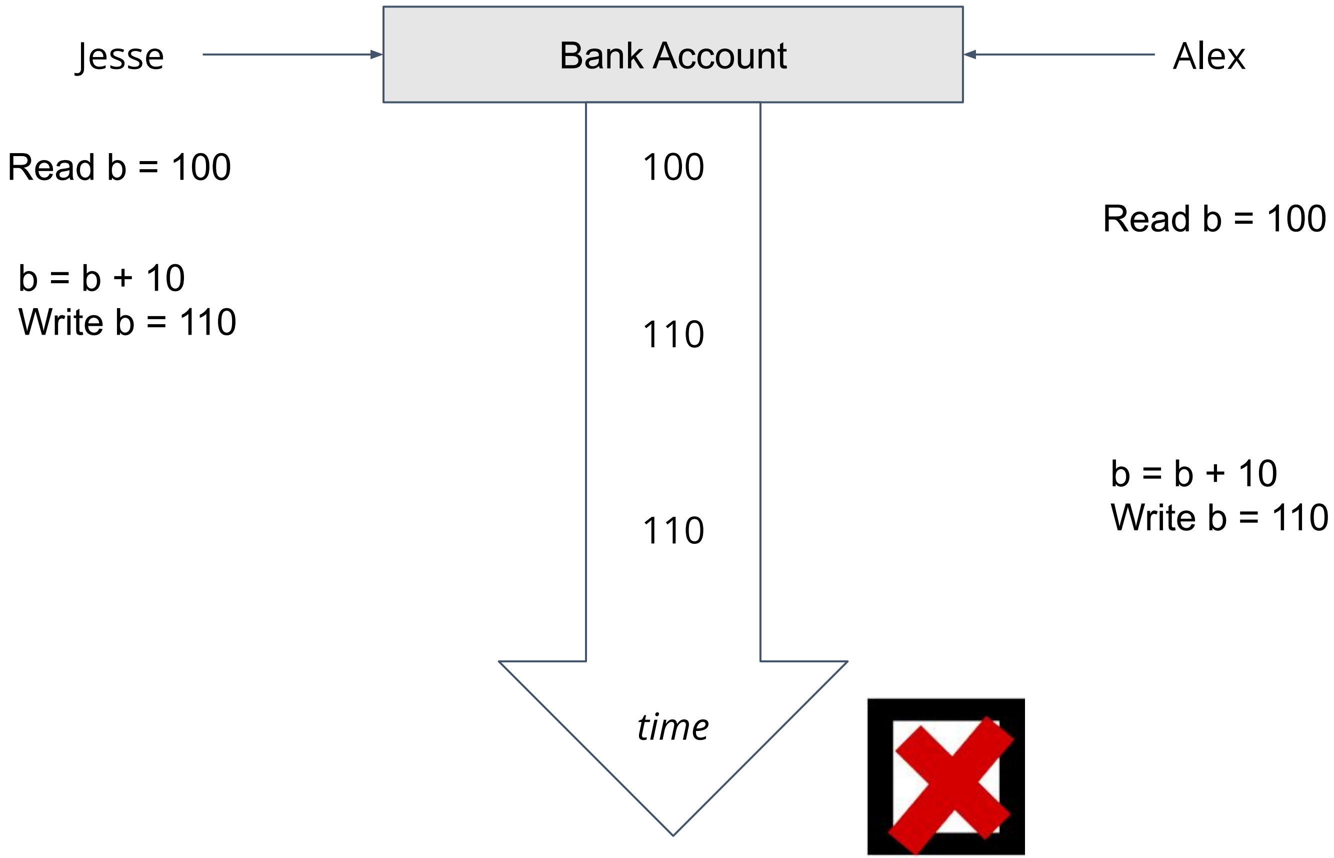


http://radonc.wikidot.com/radiation-accident-therac25

Example


Jesse
Bank Account
Alex
Read b = 100
b = b + 10
Write b = 110
100
110
120
Read b = 110
b = b + 10
Write b = 120
time

Example



Jesse
Bank Account
Alex
Read b = 100
100
Read b = 100
b = b + 10
Write b = 110
110
b = b + 10
Write b = 110
110
time

Go and Concurrency

- Goroutines
- The sync package - https://golang.org/pkg/sync
  - sync.WaitGroup
  - sync.Mutex

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

Go Runtime Scheduler

- Schedules goroutines inside an OS thread (main thread)
  - Like a little OS inside a single OS thread
- Go runs on a main thread and switches goroutines on one thread
- Go runtime scheduler uses Logical Processor which is mapped to a thread
- Typically there is one Logical Processor which is mapped to a main thread
- Since all these goroutines are running on one thread, we don't have parallelism but concurrency
- Increase number of Logical Processors - mapped to different threads and OS can map those threads to different cores
- Program can determine how many Logical Processors will be there; default is 1 (so we'll have concurrent execution of routines) but can be increased (so we might have parallel goroutines execution - if OS schedules running different threads on different cores)

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

Exercises

- Recall the exercise:
- 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 introduces some global events that every thread sees at the same time

Synchronization and Go

- `type WaitGroup`
  - `func (wg *WaitGroup) Add(delta int)`
  - `func (wg *WaitGroup) Done()`
  - `func (wg *WaitGroup) Wait()`

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

- Channels
  - See COS 418

WaitGroup

- Forces a goroutine to wait for other goroutines
- WaitGroup - a group of goroutines that a goroutine has to wait for
- A goroutine will not continue until all goroutines from WaitGroup finish
- Can wait on one or more other goroutines

- Create a WaitGroup

```
var wg sync.WaitGroup
```

- Set the size of the `WaitGroup`

```
wg.Add(num_goroutines)
```

- Pass a pointer to the WaitGroup to each go routine

```
func f(wg *sync.WaitGroup)
```

- When goroutine completes, invoke Done

```
wg.Done()
```

- Invoke Wait - blocks until all goroutines complete

```
wg.Wait()
```

WaitGroup Exercises

Consider this program:

```
func doWork(id int, sec int) {
	fmt.Printf("goroutine %d - entered. ", id)
	fmt.Printf("Sleep for %d seconds.\n", sec)
	time.Sleep(time.Duration(sec) * time.Second)
	fmt.Printf("goroutine %d - exits. ", id)
	fmt.Printf("Slept for %d seconds\n", sec)
}
func main() {
	rand.Seed(time.Now().UnixNano())
	for i := 1; i <= 5; i++ {
		go doWork(i, rand.Intn(5) + 1)
	}
	fmt.Println("Main goroutine exit")
}
```

- Run the program

  https://play.golang.org/p/nb8IJC3lyIt

- Modify the program so that each worker prints its:
  - Enter statement
  - Exit statement

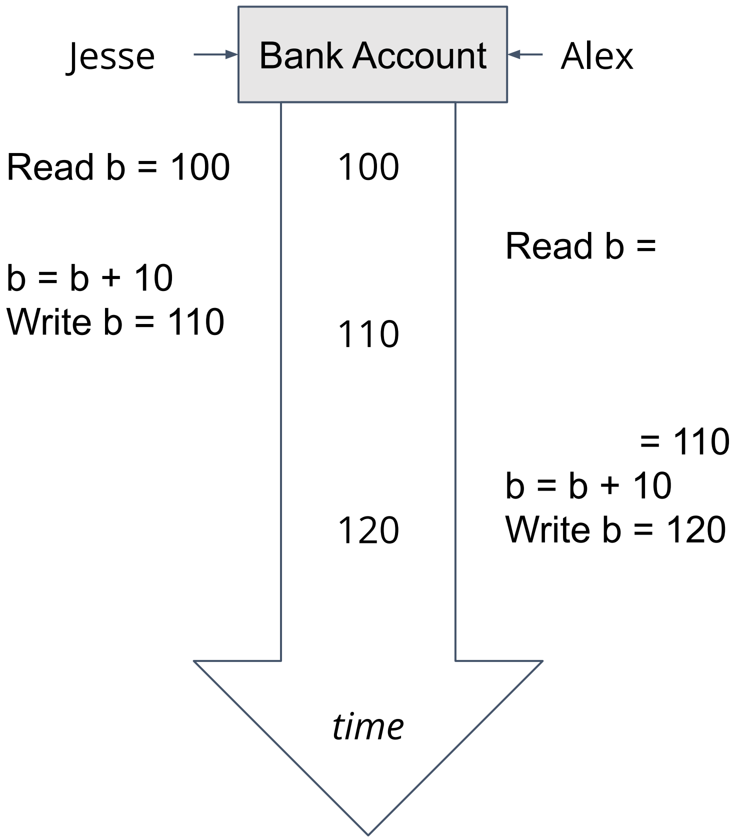
Mutex (Mutual Exclusion)

- 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)
  - Notifies others that shared variable is in use
  - 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
  - Notifies others that it is done with using shared variable
  - 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? Discuss.

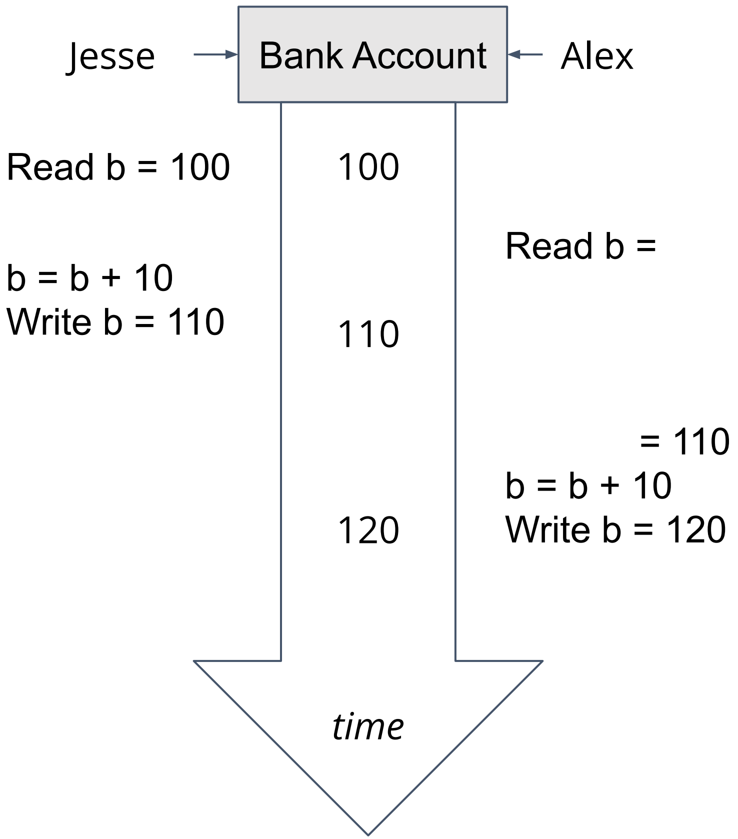
- Consider this program

  https://play.golang.org/p/c-D5UiTmgnX

- Copy this program to your local machine - build and then execute multiple times
    - Not different behavior than Go playground

- Use `Lock()` and `Unlock()` to make these programs concurrency-safe

Mutex Exercise - Bank Account



- Make this code concurrency-safe

https://play.golang.org/p/sbc5lly8cqe

Interesting Example

Consider:

```go
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
- Discuss.

Next Week - Dining Philosophers

References

Derived from:

http://www.bojankomazec.com/2019/02/concurrency-in-go-notes-on-coursera.html