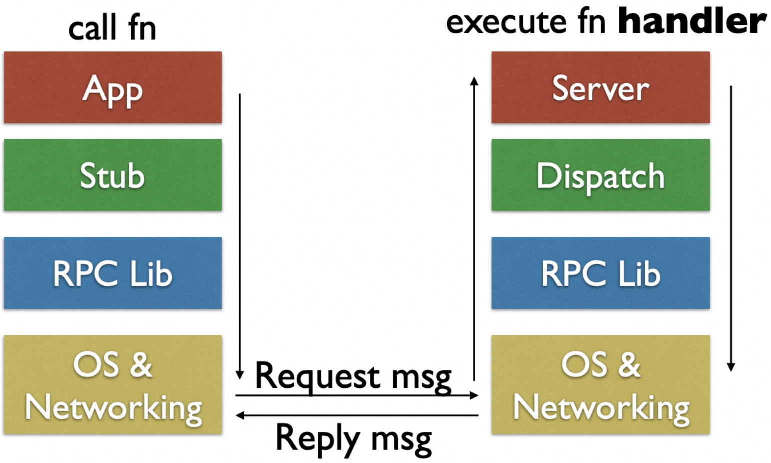
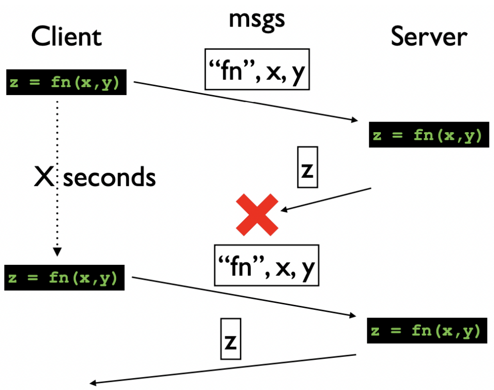
CAS CS 350
Lec 5

MapReduce

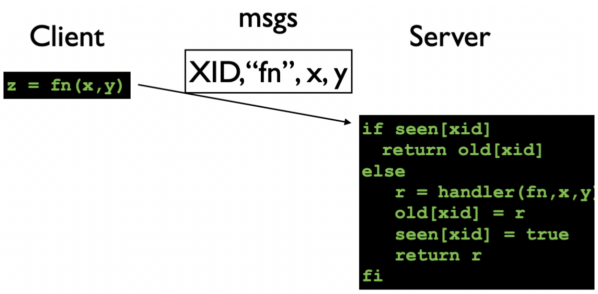
1. RPC (review)
    a. Calling function in a remote machine as if it exists on your program



    b.
    c. Stub → responsible for serializing (convert any memory object that will be sent over the wire to other machine to binary) objects
    d. Dispatch → responsible for converting binary back into memory objects
    e. At least once failure model



    f.
    g. Problem → you do not know what happened with the previous request and you might get duplicates of the same data that should be handled later
    h. At most once



```
z = fn(x,y)
```

```
if seen[xid]
  return old[xid]
else
   r = handler(fn,x,y
   old[xid] = r
   seen[xid] = true
   return r
fi
```

i.

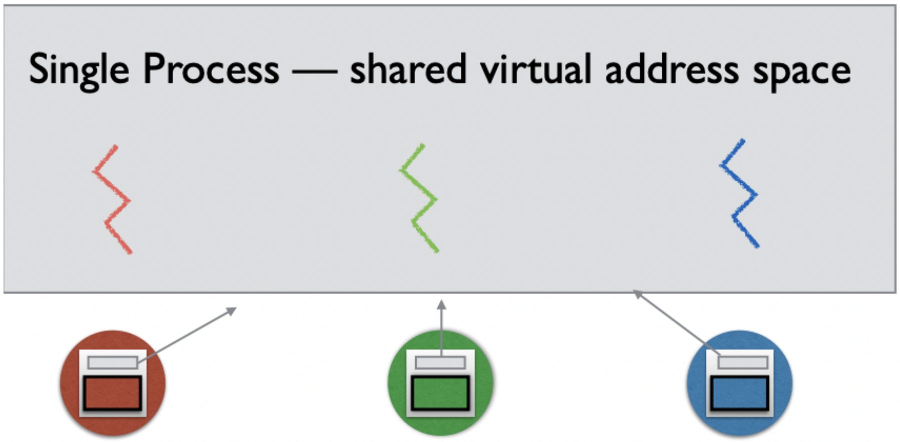
j. The unique ID, that is stored in the memory, detects duplicates (with the unique ID) and does not resend the data once the data is received by client

k. How do we delete things from old and seen?
   - i. Get an ack from the client for XID for which it has received responses
   - ii. The client includes XID in the RPC and sends the unique RPCs to servers (when the server replies with the RPC, the client acknowledges that it has received the request)

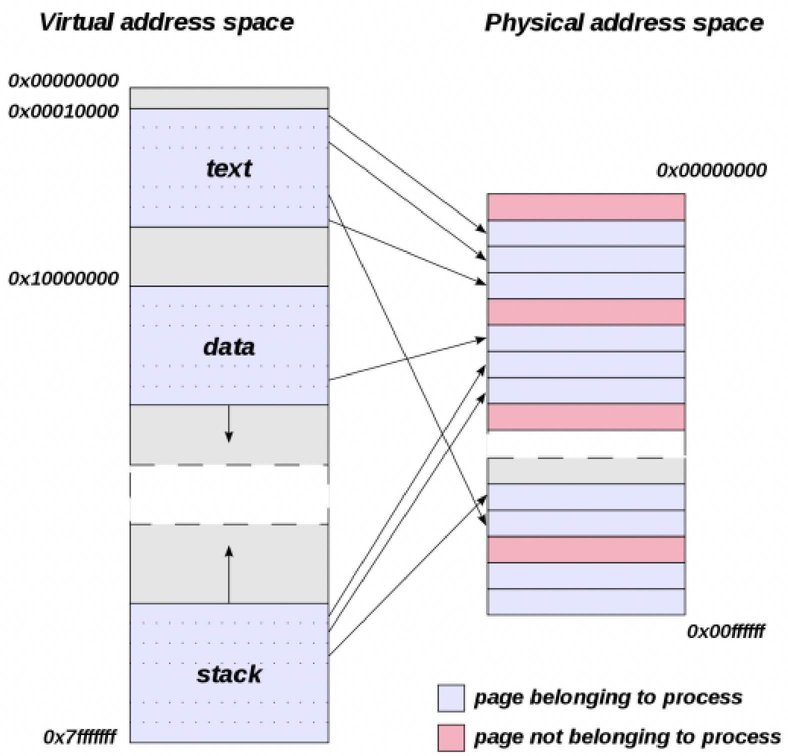
2. Some related ideas
   a. Send recent retired XID’s with next request
      - i. Have unique ID for each RPC request
      - ii. When you get the reply from the server, acknowledge that it has received the RPC request so that the server can remove all the request with that unique ID

   b. Use sequence numbers as XID
      - i. XID = <client id, seq #>
        1. ex) <42, 0>
        2. <42, 1>
      - ii. Sequence number goes up with every successful response so server knows all prior XID’s are retired
      - iii. Client contains sequence number and the sequence number increases every time the client receives RPC from the server
      - iv. Every time the client requests RPC to server, it sends the sequence number with the request so that the server knows that the client received the previous request and can delete the duplicate previous requests (If the client did not receive the request, the sequence number does not go up)

3. There is lots of subtly here
   a. There is something that you will have to think about and play with:
      - i. What happens with sequence numbers if client is allowed to make concurrent requests?
      - ii. What happens if duplicate request comes in while the original is still executing
      - iii. What happens if server crashes and is restarted the face of duplicate requests?

4. How about Once (the most ideal case) → hard to achieve in practice
    - a. Need at-most once with at least once unbounded retries
    - b. And fault tolerant server implementation
5. Go RPC
    - a. Includes at most once library → will never see a duplicate in the same server
    - b. At most once with respect to a single client server
    - c. Built on top of single TCP connection (relies on TCP protocol (message arrives in order with no duplicates) of SYN-ACK)
        - i. Thus TCP handles retries and duplicates under the covers
    - d. Returns error if reply is not received
        - i. Eg. connection broken (TCP timeout)
6. Not Good Enough in general
    - a. Lab 1
        - i. GO RPC will avoid single worker from ever seeing a duplicate request from the coordinator
        - ii. But if the coordinator doesn’t get a response from a worker and assign the same task to another worker, it may end up with duplicates
        - iii. Go RPC can’t detect this
7. Threads (go routine0
    - a. Basic Idea
        - i. 

          Single Process — shared virtual address space
        - ii. Each thread belongs to process and each process has at least one thread
8. Virtual vs Physical Memory



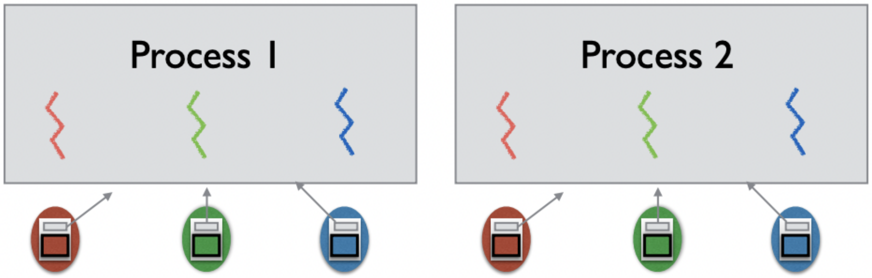
a.

b. The threads see contiguous space

c. In reality, the right is the physical memory (the addresses are not always next to each other but are spread across)

d. The thread does not know anything about the physical address space (application does not care and the operation system has to handle it)

9. Each process has its own virtual address space



a.

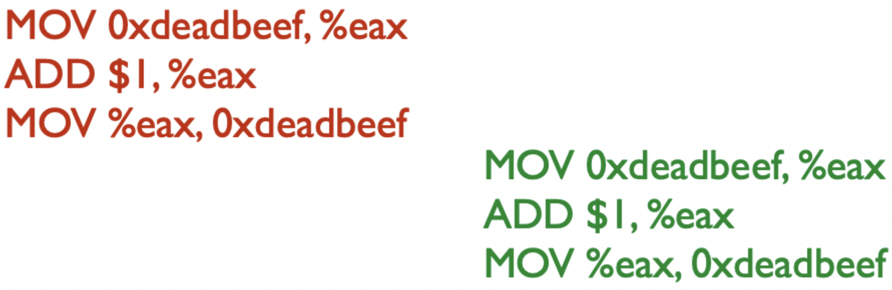
b. How to exchange data between process

   i. You cannot just pass a pointer (due to different memory space) since the address of virtual and physical is different (each virtual address maps to a different physical address in one memory so simply pointer would not map the same physical address despite having the same virtual address)

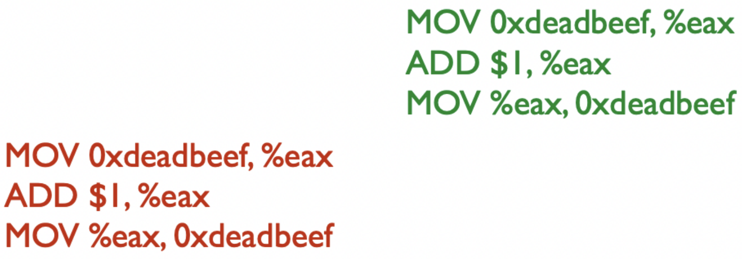
   ii. Method 1: Sockets → used to exchange data in processes

    iii. Method 2: Transfer message through RPC
        1. Relationship between RPC and socket : RPC uses sockets to communicate (in Go's case, it is TCP), RPC is one layer above of sockets
        2. Whenever we do this step, we need to go through the steps of serialization (because the two processes do not see the same physical memory)

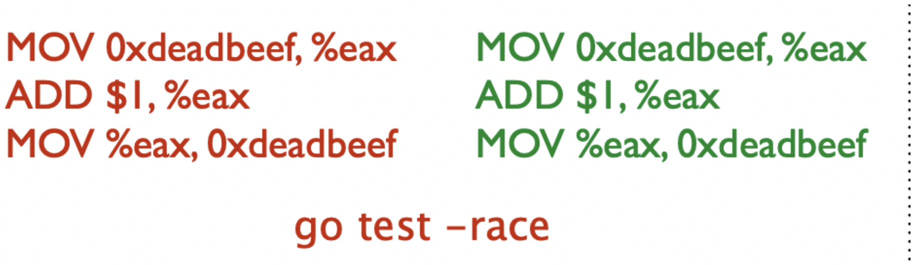
10. Basic problem : r/w shared data structures concurrently

    a. 

```
MOV 0xdeadbeef, %eax
ADD $1, %eax
MOV %eax, 0xdeadbeef
                         MOV 0xdeadbeef, %eax
                         ADD $1, %eax
                         MOV %eax, 0xdeadbeef
```

    b. Thread on the left goes first
    c. 

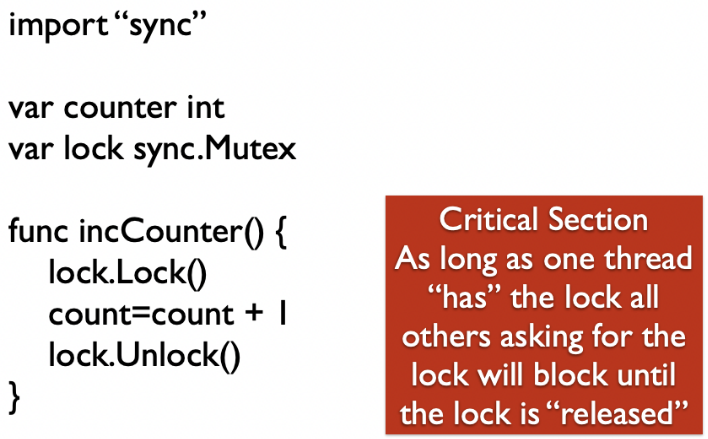
```
                    MOV 0xdeadbeef, %eax
                    ADD $1, %eax
                    MOV %eax, 0xdeadbeef
MOV 0xdeadbeef, %eax
ADD $1, %eax
MOV %eax, 0xdeadbeef
```

    d. Thread on the right goes first
    e. 

```
MOV 0xdeadbeef, %eax      MOV 0xdeadbeef, %eax
ADD $1, %eax              ADD $1, %eax
MOV %eax, 0xdeadbeef      MOV %eax, 0xdeadbeef

              go test -race
```

    f. Both threads go at the same time → problems can occur (behavior can be undefined → memory gets corrupted)
    g. In this case, we need to coordinate the threads
    h. Without it, we have data race
        i. How to avoid data race → use Mutual Exclusion and Go Mutex aka LOCK



```
import "sync"

var counter int
var lock sync.Mutex

func incCounter() {
    lock.Lock()
    count=count + 1
    lock.Unlock()
}
```

Critical Section
As long as one thread "has" the lock all others asking for the lock will block until the lock is "released"

ii.

iii. Mark critical section → lock and unlock

iv. Whatever is inside the critical section is accessed by only one thread at a time

v. All threads must undergo lock and unlock

vi. Without passing, there will be errors detected

i. Go has data race detector that is useful in assignments → go test -race (Go lets us know but Go does not solve the problem for us)

11.