Lec 2: RPC and Threads

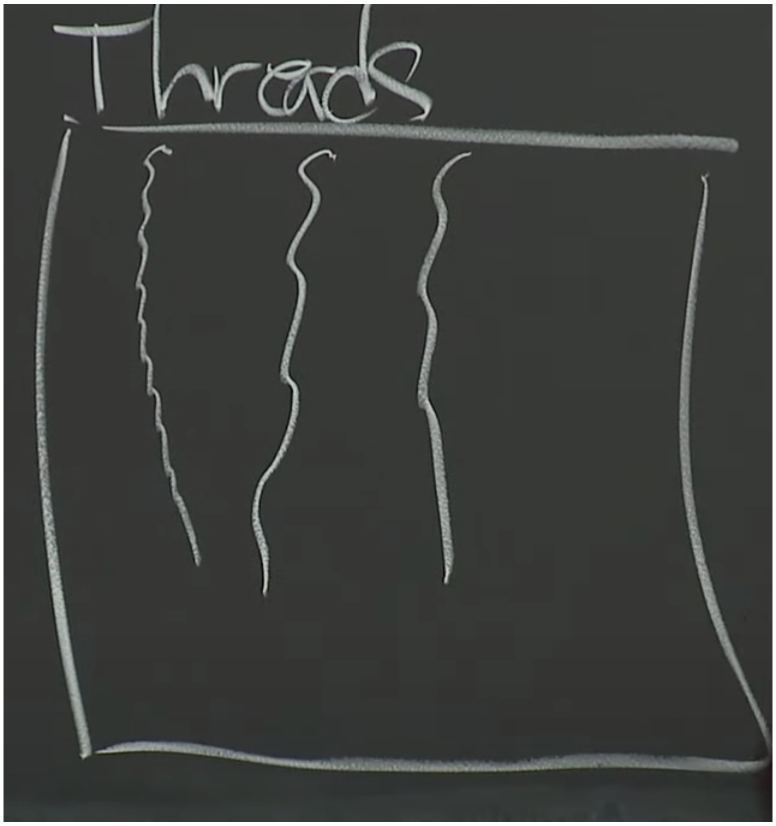
Tags	

Go

- good support for concurrency stuff
- convenient RPC package
- type safe, memory safe
- garbage collection

Threads

- main tool to manage concurrency in program
- one program to talk to a bunch of computers



The rectangle is the address space, the squiggly lines are the threads

- separate program counter, stack, registers for stack for thread control
- I/O Concurrency
  - create one thread for each RPC call, and wait for the results
- parallelism
  - multi-core machine, use CPU resources from all the cores
- convenience

Aside: concurrent programming vs asynchronous programming (event-driven programming)

- single thread, single loop, wait for any event that might trigger processing
- threads are more convenient
- CPU parallelism could be achieved from threads

- threads can be expensive if there are many (e.g. many clients), even-driven programming could be less costly in this case

Aside: Threads vs Processes

- process is an executing program, is itself a single address space, inside the program having many threads
- in traditional structure, between processes, there is no communication such as mutex, sync with channels,

Aside: When a context switch happens, does it happen for all threads

- the OS pick which big thread to run and within that process go may have a choice of go routines to run, not necessarily picking threads from the same process in a multi-core machine

Thread Challenges

- shared data, race (condition)
  - e.g. global counter n, n = n + 1
  - machine code
    - LD x, r
    - Add 1, r
    - ST r, x // one thread must finish the store in order to let other threads see the updated value
- coordination
  - we sometimes want different threads to interact
  - channels
  - sync.cond
  - wait group

        - launch a number of goroutines
- deadlock
  - T1 waits for T2, and T2 waits for T1

Web Crawler Example

- avoid cycles
- sometimes take long time to fetch pages
- know when the crawl is finished, the hardest part

```
package main

import (
  "fmt"
  "sync"
)
//
// Several solutions to the crawler exercise from the Go tutorial
// https://tour.golang.org/concurrency/10
//

//
// Serial crawler
//
// a map is a pointer, built in the language
func Serial(url string, fetcher Fetcher, fetched map[string]bool) {
  if fetched[url] {
    return
  }
  fetched[url] = true
  urls, err := fetcher.Fetch(url)
  if err != nil {
    return
  }
  for _, u := range urls { // essentially a dfs
    Serial(u, fetcher, fetched)
  }
  return
}
```

```
//
// Concurrent crawler with shared state and Mutex
```

```
//

type fetchState struct {
  mu      sync.Mutex
  fetched map[string]bool
}

func ConcurrentMutex(url string, fetcher Fetcher, f *fetchState) {
  f.mu.Lock()
  already := f.fetched[url] // shared by all threads
  f.fetched[url] = true
  f.mu.Unlock()

  if already {
    return
  }

  urls, err := fetcher.Fetch(url)
  if err != nil {
    return
  }
  var done sync.WaitGroup // counter
  for _, u := range urls {
    done.Add(1) // increment counter
    //u2 := u
    //go func() {
    // defer done.Done()
    // ConcurrentMutex(u2, fetcher, f)
    //}()
    go func(u string) { // go routinue,
// u is needed here because of for-loop change u to point to
// a different string in every iteration
      defer done.Done() // decrement counter
      // call done before the surrounding function finishes, always call
      ConcurrentMutex(u, fetcher, f)
    }(u)
  }
  done.Wait()
  return
}

func makeState() *fetchState {
  f := &fetchState{}
  f.fetched = make(map[string]bool)
  return f
}
```

```
//
// Concurrent crawler with channels
//
```

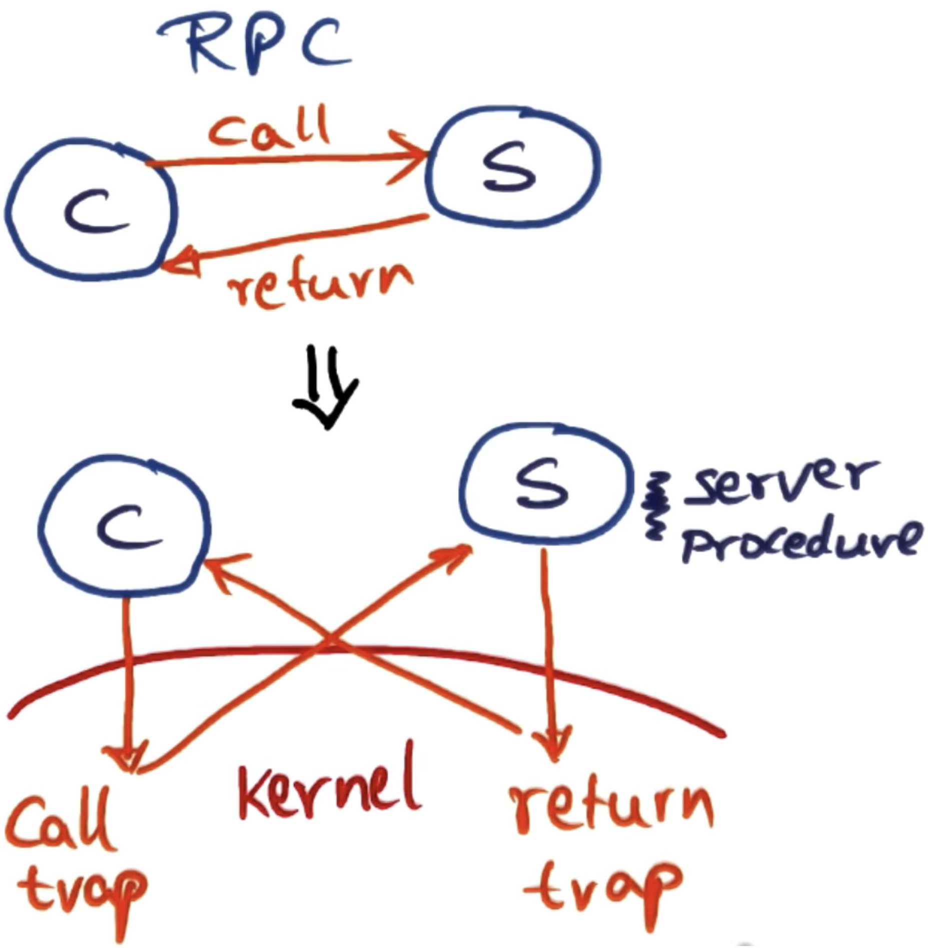
```go
func worker(url string, ch chan []string, fetcher Fetcher) {
  urls, err := fetcher.Fetch(url)
  if err != nil {
    ch <- []string{}
  } else {
    ch <- urls
  }
}

func coordinator(ch chan []string, fetcher Fetcher) {
  n := 1
  fetched := make(map[string]bool)
  for urls := range ch {
    for _, u := range urls {
      if fetched[u] == false {
        fetched[u] = true
        n += 1
        go worker(u, ch, fetcher)
      }
    }
    n -= 1
    if n == 0 {
      break
    }
  }
}

func ConcurrentChannel(url string, fetcher Fetcher) {
  ch := make(chan []string)
  go func() {
    ch <- []string{url}
  }()
  coordinator(ch, fetcher)
}
```

- -race flag in go can detect the race condition
  - run-time analysis, not static
- No upper-bound with the number of threads

Aside: What is RPC?



- IPC: processes residing in different systems
- RPC is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details
- message-based communication (message passing)
  - messages are well structured and are no longer packets of data
  - message is addressed to an RPC daemon listening to a port on the remote system
    - daemon: program that is always listening
  - each contains an identifier of the function to execute and the parameters to pass to that function
  - function is executed and any output is sent back to the requester in a separate message