Go Programming Language

Looping in parallel & Multiplexing with select

Department of Computer Science and Engineering

Go Programming Language

Looping in parallel

- Problems that consist entirely of sub-problems that are completely independent of each other are described as **embarrassingly parallel**.
- Embarrassingly parallel problems are the easiest kind to implement concurrently and enjoy performance that scales linearly with the amount of parallelism.
- Looping in parallel is implemented as go routines

Go Programming Language

Example – Web Crawler

```
func main() {
    worklist := make(chan []string)

    // Start with the command-line arguments.
    go func() { worklist <- os.Args[1:] }()

    // Crawl the web concurrently.
    seen := make(map[string]bool)
    for list := range worklist {
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                go func(link string) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

```
func crawl(url string) []string {
    fmt.Println(url)
    list, err := links.Extract(url)
    if err != nil {
        log.Print(err)
    }
    return list
}
```

Example – Web Crawler – limiting parallelism

- We can limit parallelism using a buffered channel of capacity **n** to model a concurrency primitive called a counting semaphore.
- Conceptually, each of the n vacant slots in the channel buffer represents a token entitling the holder to proceed.
- Sending a value into the channel acquires a token, and receiving a value from the channel releases a token, creating a new vacant slot.
- This ensures that at most n sends can occur without an intervening receive

Go Programming Language

Example – Web Crawler – limiting parallelism

```
func main() {
    worklist := make(chan []string)
    var n int // number of pending sends to worklist

    // Start with the command-line arguments.
    n++
    go func() { worklist <- os.Args[1:] }()

    // Crawl the web concurrently.
    seen := make(map[string]bool)
    for ; n > 0; n-- {
        list := <-worklist
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                n++
                go func(link string) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

```
// tokens is a counting semaphore used to
// enforce a limit of 20 concurrent requests.
var tokens = make(chan struct{}, 20)

func crawl(url string) []string {
    fmt.Println(url)
    tokens <- struct{}{} // acquire a token
    list, err := links.Extract(url)
    <-tokens // release the token

    if err != nil {
        log.Print(err)
    }
    return list
}
```

Go Programming Language

Multiplexing with select

- One of the most powerful tools for managing multiple channels is the **select** statement
- The select statement provides another way to handle multiple channels.
- It is like a switch statement, but for channels
- A select waits until a communication for some case is ready to proceed.
- It then performs that communication and executes the case's associated statements
- The other communications do not happen.
- A select with no cases, select{}, waits forever.

Go Programming Language

Multiplexing with select

```
func main() {
    // ...create abort channel...

    fmt.Println("Commencing countdown.  Press return to abort.")
    select {
    case <-time.After(10 * time.Second):
        // Do nothing.
    case <-abort:
        fmt.Println("Launch aborted!")
        return
    }
    launch()
}


abort := make(chan struct{})
go func() {
    os.Stdin.Read(make([]byte, 1)) // read a single byte
    abort <- struct{}{}
}()
```

Go Programming Language

Cancellation

- Sometimes we need to instruct a goroutine to stop what it is doing
- There is no way for one goroutine to terminate another directly, since that would leave all its shared variables in undefined states.
- What if we need to cancel two goroutines, or an arbitrary number?
- In general, it’s hard to know how many goroutines are working at any given moment.
- For cancellation, what we need is a reliable mechanism to broadcast an event over a channel so that many goroutines can see it as it occurs and can later see that it has occurred.
- First, we create a cancellation channel on which no values are ever sent, but whose closure indicates that it is time for the program to stop what it is doing.
- We also define a utility function, cancelled, that checks or polls the cancellation state at the instant it is called.

THANK YOU

Suresh Jamadagni

Department of Computer Science and Engineering