Programming Paradigms CSI2120 – Winter 2018

Jochen Lang

EECS, University of Ottawa

Canada

uOttawa

L'Université canadienne
Canada's university

System Programming: Go

- Concurrency and Parallelism
- Goroutines
- Channels

Concurrent Programming

- **An application is a process running on a machine**
  - A process is an independently executing entity that runs in its own address space.
- **A process is composed of one or more operating system threads**
  - Threads are concurrently executing entities that share the same address space.

Execution Thread

- **An execution thread is a sequence of executable statements and it may or may not interact with other threads**
  – Threads often share some variables (or resources) with other threads
  – Threads often (but not necessarily) have a limited life-time within a program execution
- **A thread can be blocked:**
  – Accessing a shared variable or resource used currently by another thread
  – Waiting for the result or completion of another thread
- **An application program can often be divided into a process and potentially many threads**

Parallel Programming vs. Concurrent Programming

- **Concurrent programming means expressing program structure such that is organized into independently executing actions**
  - Concurrent programming can also target a single processor
    - The processes or threads then run in turn over time according to a schedule
- **If processes or threads are running on different processors or cores simultaneously, we have a parallel program**
- **If a program is executed on multiple machines with loose interaction, it is commonly called distributed programming**
- **If a program is executed on a graphics card or a tightly integrated cluster, it is often called massively parallel programming**

Concurrent Programming Languages

- **A concurrent programming language must support:**
  - the creation and execution of processes and threads
  - synchronization operations
    - Cooperative synchronization: A process waits for the execution of another before continuing its own execution.
      - deterministic concurrency
    - Competitive synchronization: Multiple processes use the same resource with some form of locking mechanism for mutual exclusion.
      - non-deterministic concurrency
  - data communication between threads and processes
    - may use mechanisms using inter-process communication defined by the operating system

Reminder: Threads in Java

- Option implementing interface runnable

```
public class HelloRunnable
  implements Runnable {

  public void run() {
    System.out.println(
     "Running in a
      thread");
  }

  public static void
    main( String args[]) {
   (new Thread( new
   HelloRunnable())).start();
  }
}
```

- Option subclassing Thread

```
public class HelloThread
  extends Thread {

  public void run() {
    System.out.println(
      "Running in a
       thread!");
    }

  public static void
    main(String args[]) {
    (new
   HelloThread()).start();
    }
}
```

Level of Concurrency

- **At the (set of) statement level:**
  - Sets of statements are executed concurrently while the main process suspends its execution. All threads share the same data (OpenMP)
- **At the sub-program level:**
  - New process is created for running a subroutine. Once the new process starts, the calling process continues its execution. Requires a synchronization mechanism.
- **At the object level:**
  - Each object instance of a class competes for resources and methods on different objects run concurrently. Class variables (attributes) are not shared.
- **At the program level:**
  - Parent process run one or more child processes. Child process Id must be known by parent. Data may be shared.

Type of Concurrency

- **Physical**
  - Multiple processes/threads share different processors or cores
- **Logical**
  - Multiple processes/threads share execution time on a single processor.
- **Distributed**
  - Processes/threads of an application share several machines over a network

Concurrency in Go

- **Two mechanisms are provided**
  - Non-deterministic Concurrency
    - More traditional threads with low-level synchronization
    - Mutex in synch package
    - Use is discouraged in Go
  - Deterministic Concurrency
    - Communicating Sequential Processes (CSP)
    - Based on message passing between threads
    - goroutines and channels
    - Recommended approach

Concurrency in Go

- **CSP**
  - Based on the idea that avoiding data sharing will avoid the biggest problem in concurrent programming
  - Threads in CSP do not communicate by sharing data
  - Rather they share data by communicating
  - Timing of threads is based on messaging between threads

Goroutines

- **Parts of a Go program that run concurrently are organized in *goroutines***
  - *goroutines* can run in several threads
  - several *goroutines* can run in one thread
    - no 1:1 correspondence to OS threads
- ***goroutines* run in the same address space**
  - shared memory access can be used but is discouraged
- ***goroutines* are designed to be light-weight**
  - inexpensive to create
  - automatic (segmented) stack management
- **A *goroutine* can be implemented as a function or method**
- **A *goroutine* is invoked using the keyword go**

Calling a *Goroutine*

- ***Goroutines* are functions or methods called with go**
  - The default number of OS threads is one for all goroutines
  - Can be changed with the environment variable GOMAXPROCS
  - Can also be changed with the runtime package

```
import "runtime"
func main() {
    // change max number of OS threads to 3
    runtime.GOMAXPROCS(3)
    …
    go foo()
    …
}
```

Example: Calling *goroutines*

```
func main() {
    runtime.GOMAXPROCS(3)
    sTime := time.Now(); // time it
    fmt.Println("Starting")
    go letters() // goroutine A
    go numbers() // goroutine B
    fmt.Println("Waiting …")
    time.Sleep(2*time.Second)
    fmt.Println("\nDone\n")
    eTime := time.Now();
    fmt.Printf("Run time: %s", eTime.Sub(sTime))
}
```

Example: *goroutines*

```
func numbers() {
    for number := 1; number < 27; number++ {
       // pause before every print
       time.Sleep(10*time.Millisecond)
       fmt.Printf("%d ", number)
    }
}

func letters() {
    for char := 'a'; char < 'a'+26; char++ {
       time.Sleep(10)
       fmt.Printf("%c ", char)
    }
}
```

Example Execution

```
Starting
Waiting ...
a b c d e f g h i j 1 k l m n o p q r s t 2 u v w
x y z 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26
Done
Run time: 2s
```

- **Running it without *goroutines* (sequential)**

```
Starting
a b c d e f g h i j k l m n o p q r s t u v w x y
z 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 Waiting ...
Done
Run time: 2.286s
```

Communication between *goroutines*

- ***goroutines*** **are designed to effectively communicate by message passing for**
  - the exchange of information
  - for synchronizing the execution
- **Implemented in Go through channels**
  - channels are similar to pipes

Concept of *channels*

- **Data is passed around through channels**
  - only one *goroutine* has access to a channel at any given time
  - communication is synchronized by design
    - no race conditions can occur
- **Ownership of the data is passed around as well**
- **A channel is a data queue (FIFO)**
- **Channels are typed (only one data type can be passed around on a channel)**

Channel Declaration

- **Declaring a channel only creates a reference**
  - Use make to allocate space for the channel
  - Example: Channel for strings

    ```
    var ch chan string
    ch = make(chan string)
    ```

  - Or shorter with initializer declaration

    ```
    ch := make(chan string)
    ```

  - By default, a channel has a capacity of 1 element
  - Channels are first-class object
  - To send or receive, use the arrow operator

    ```
    ch <- str // send
    str = <- ch // receive
    ```

Example: Communicating *goroutines* through channels

```
func main() {
   ch := make(chan string)
   go sendString(ch)
   go recieveString(ch)
   time.Sleep(1*time.Second)
}

func sendString(ch chan string) {
   ch <- "Ottawa"
   ch <- "Toronto"
   ch <- "Gatineau"
   ch <- "Casselman"
}

func recieveString (ch chan string) {
   var str string
   for {
        str= <-ch
        fmt.Printf("%s ", str)
   }
}
```

Range Loop applied to a Channel

- We can loop over the incoming elements
- Loops over the channel until it is closed
- Example uses a lambda as go routine

```
func recieveString(ch chan string) {
  go func() {
   for str := range ch {
        fmt.Printf("%s ", str)
   }
  }()
}
```

Closing channels

- **Channels may be explicitly closed**

```
func sendString(strArr []string ) chan string {
   ch := make(chan string)
   go func() { // start a lambda in a go routine
     for _,s := range strArr {
            ch <- s
     }
     close(ch)
   }()
   return ch
}
```

- **We can test if the channel has been closed**

```
for {
     str, ok := <- ch
     if !ok {
            break;
     }
     fmt.Printf("%s ", str)
}
```

Synchronization across Channels

- We can use select for non-blocking channel i/o. Similar to a switch but depending on which channel receives or ready to send an element

```
Forever:
    for {
            select {
            case str, ok := <- ch1:
                    if !ok {
                            break Forever
                    }
                    fmt.Printf("%s \n", str )
            case str, ok := <- ch2:
                    if !ok {
                            break Forever
                    }
                    fmt.Printf("%s \n", str )
            }
    }
```

Timers in Go

- **A timer in Go has a channel**
- **We can check its status or read from its channel**

```
    outOfHere := time.NewTimer(time.Second * 2)
Forever:
    for {
        select {
            … \\ Omitted
            case <-outOfHere.C:
                fmt.Printf("Time is up\n" )
                break Forever
        }
    }
```

Summary

- Concurrency and Parallelism
- Goroutines
- Channels
- Channel Synchronization
- Timer