Threads and Concurrency

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

Distributed Systems
ECE419

Overview

- Go programming language
- Threads and synchronization
- Web crawler in Go

Why Go for labs?

- Designed for building distributed infrastructure services
    - E.g., Kubernetes, a container deployment system
- Good support for threads, RPC
- Other reasons
    - Built-in strings, hash maps, dynamic arrays (easier to program)
    - Statically compiled (relatively fast)
    - Type-safe, memory-safe (less bugs)
    - Garbage-collected (no use-after-free problems)
    - Good libraries, deployment toolchain

Hello world in Go

```go
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.ListenAndServe("localhost:8080",
        http.HandlerFunc(hello))
}

func hello(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "hello, world\n")
}
```

Go resources

- Short paper describing motivation for and design of Go
  - https://cacm.acm.org/research/the-go-programming-language-and-environment/
- Getting familiar with Go
  - Start with the tour of Go: http://tour.golang.org/
  - After that: https://golang.org/doc/effective_go.html

Threads and Synchronization

What are threads?

- Programs use threads to do multiple things at once
  - e.g., a video player needs to download and display video, it can use two threads, one for each operation
- A thread executes a stream of instructions serially, like a non-threaded program
- Threads share memory, e.g., variables
- Each thread has some per-thread state: program counter, registers, stack

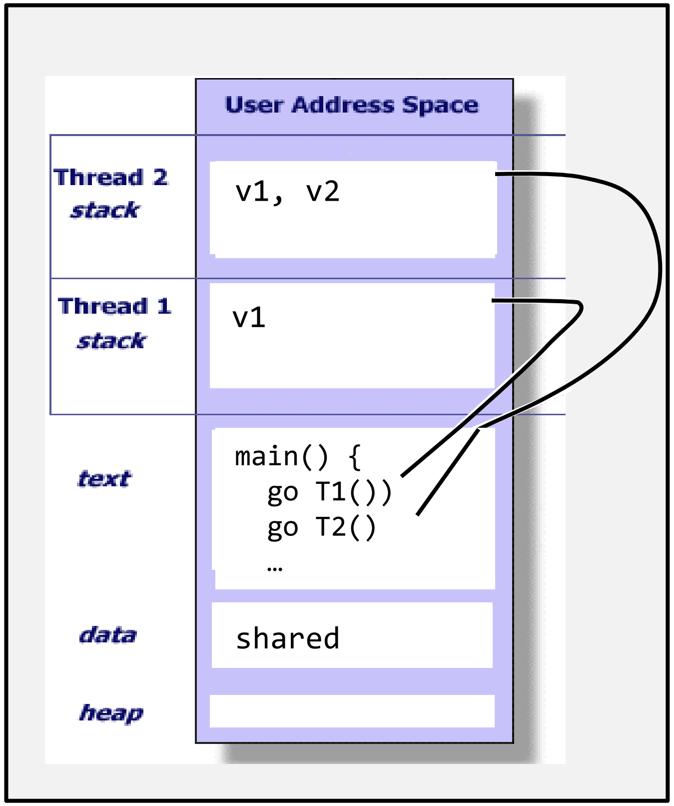
A simple threaded program

- Program has three threads: main, T1 and T2

Threads communicate by accessing shared data

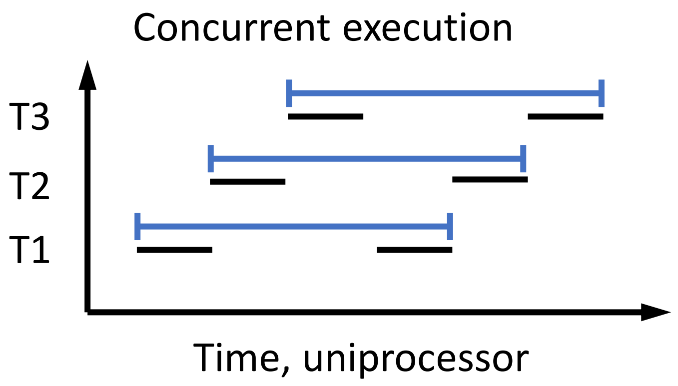
```
var shared int

main() {
  go T1()
  go T2()
  for {
  }
}
```

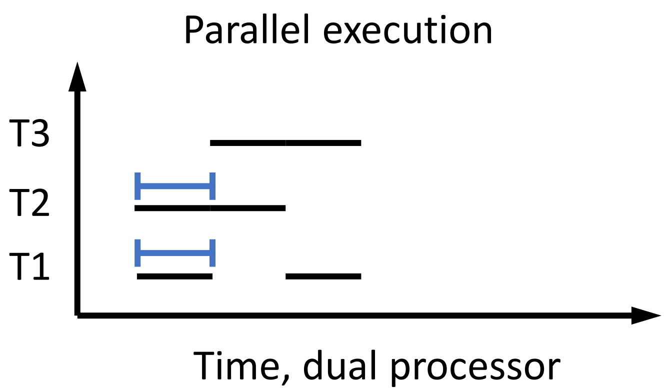
```
T1() {
  v1 := 1
  shared = v1
}

T2() {
  v1 := shared
  v2 := 2
}
```



Concurrency vs. parallelism

- Concurrent execution:
  - Threads execute in overlapping time intervals
  - Allows a program to use CPU and IO devices in parallel
- Parallel execution
  - Threads execute at the same time
  - Allows a program to use multiple CPUs in parallel





Why use threads?

- Enable both IO concurrency and CPU parallelism
- I/O concurrency
  - Clients send requests to a server in parallel, wait for replies
  - Server processes many simultaneous client requests
  - Server uses a thread per request
  - A thread waits while reading data from slow disk for client X
  - Another thread continues processing a request from client Y
- CPU parallelism
  - Two threads execute a computation in parallel on two cores

Threading challenges

- Race conditions
  - Certain thread interleavings cause incorrect behavior
  - E.g., two threads update same shared variable, n = n+1, update can be lost since it is not atomic
  - Solution: use locks to perform update in a critical section
- Synchronization
  - Some operations need to be performed in a particular order

```
T1() {
  v1 := 1
  shared = v1
}
```

initialize

Synchronize →

```
T2() {
  v1 := shared
  v2 := 2
}
```

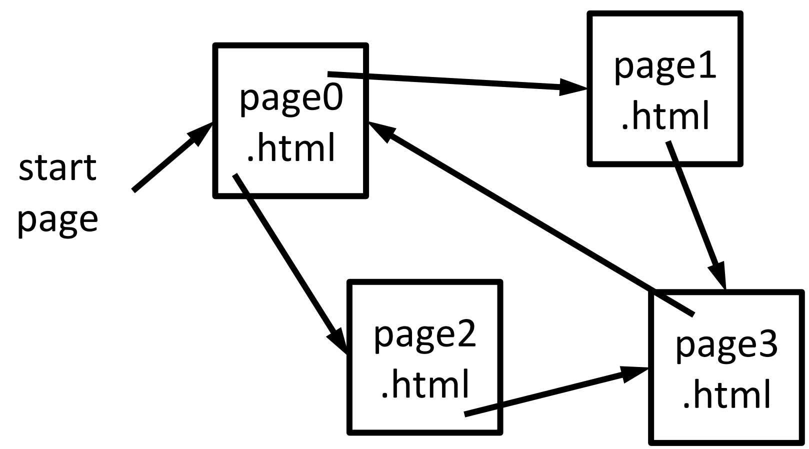
read

  - Solution: use condition vars, semaphores, bounded buffer

Webcrawler in Go

Webcrawler

- Webcrawler fetches web pages
  - Starts at a page
  - Parses links (URLs) in page and follows those links recursively
  - Page contents and their URLs can be sent to an indexer



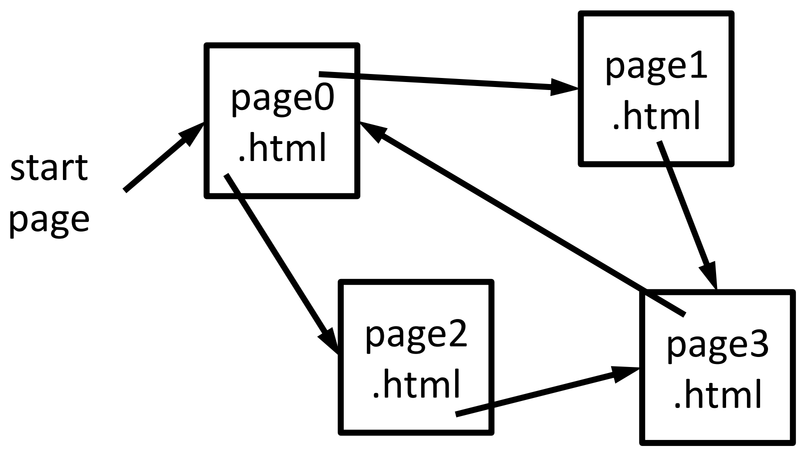
Webcrawler challenges

- Fetch each page once
  - Minimize network bandwidth
  - Avoids getting stuck in cycles

  ⇒ Need to remember URLs visited

- Fetch pages efficiently
  - Network latency is more limiting than bandwidth
  - Fetch pages in parallel

  => Use threads for concurrency



Webcrawler implementation in Go

- Let’s look at three implementations
  - Serial
  - Concurrent, synchronization using shared data and locks
  - Concurrent, synchronization using bounded buffer (channels)

Shared data or channels

- Most problems can be solved in either style
- For synchronization (waiting/notification) use channels, sync.Cond, Sleep, etc.
- For state that cannot be easily moved between threads (e.g., large data structures), use shared state and locks

Conclusions

- Threads allow a program to utilize resources efficiently
  - Allow CPU and IO devices to run concurrently
  - Allow using multiple CPUs in parallel
- Threaded programs must handle
  - Races when threads access shared data concurrently
    - Solution: use mutual exclusion
  - Synchronization when ordering needed across threads
    - Solution: use condition variables, channels