Parallel Programming

I/O and concurrency

Morgan Ericsson

Today

- Concurrency and I/O
- HTTP requests
- HTTP servers
- REST APIs

Concurrency and I/O?

- So far, focus on parallelism
- Remember I/O vs CPU bound
- Another reason to use concurrency is to “hide” I/O wait
  - We simply do other things while we wait for I/O to finish

Concurrency and I/O?

Device	CPU cycles	Proportional
L1 cache	3	3 sec
L2 cache	14	14 sec
RAM	250	250 sec
disk	41 000 000	1.3 years
network	240 000 000	7.6 years

An example

Example settings

- A typical example of I/O bound tasks is networking and web APIs
- We will use the Nobel prize REST API
- Information about prizes and laurates

Example API call

```python
import requests
import json

BU = 'http://api.nobelprize.org/2.1/'
PU = 'nobelPrize/'
LU = 'laureate/'

r = requests.get(f'{BU}{PU}phy/2022')
if r.ok:
  for np in r.json():
    for ll in np['laureates']:
      print(ll['fullName']['en'])
```

HTTP Request

- HTTP is the hypertext transfer protocol
- Simple protocol to transfer text
  - GET, PUT, POST, ...
- Used by web servers
- And to develop API
  - More about this in a later course

HTTP Request

```
GET /clock HTTP/1.1
Host: localhost:3000
User-Agent: curl/7.81.0
Accept: */*
```

HTTP Response

```
HTTP/1.1 200 OK
Date: Mon, 26 Feb 2024 12:40:49 GMT
Content-Length: 29
Content-Type: text/plain; charset=utf-8

12:40:49 UTC
```

HTTP Request

```
POST /area HTTP/1.1
Host: localhost:3000
User-Agent: curl/7.81.0
Accept: */*
Content-Type: application/json
Content-Length: 24

{"height":30,"width":20}
```

HTTP Response

```
HTTP/1.1 200 OK
Date: Mon, 26 Feb 2024 12:53:20 GMT
Content-Length: 12
Content-Type: text/plain; charset=utf-8

{"area":600}
```

JSON

```
[
  {"id":"940",
  "knownName":{
    "en":"Michael W. Young",
    "se":"Michael W. Young"
  },
  "gender":"male",
  "birth":{
    "date":"1949-03-28",
    "place":{
      "city":{"en":"Miami, FL",
      "no":"Miami, FL",
      "se":"Miami, FL"
    },
    "country":{
      "en":"USA",
      "no":"USA",
      "se":"USA"
    }
    ...
  }
```

JSON

- JavaScript Object Notation
- Text-based interchange format
- Supports some types: number, string, array, boolean, object
- Language-independent, most languages has support for json
  - e.g. `json` in Python or `encodings/json` in Go

Any (unknown) type

```go
var a any
a = "Hello"
b := string(a)
fmt.Println(b)
```

```
./prog.go:12:14: cannot convert a (variable of type any) to type string: need type assertion
```

Type assertions in Go

```go
var a any
a = "Hello"
b := a.(string)
fmt.Println(b)
```

Type assertions in Go

```go
var a any
a = "Hello"
b := a.(int)
fmt.Println(b)
```

```
panic: interface conversion: interface {} is string, not int
```

Example API call

```go
url := "https://api.nobelprize.org/2.1/nobelPrize/phy/2022"

resp, err := http.Get(url)
if err != nil {
  log.Fatal(err)
}
defer resp.Body.Close()

if resp.StatusCode != http.StatusOK {
  log.Fatal(resp.StatusCode)
}

data, err := io.ReadAll(resp.Body)
if err != nil {
  log.Fatal(err)
}
```

Example API call

```go
var result []map[string]any
json.Unmarshal(data, &result)

tmp := result[0]["laureates"].([]any)
for _, k := range tmp {
  fn := k.(map[string]any)["fullName"]
  en := fn.(map[string]any)["en"]
  fmt.Println(en.(string))
}
```

Why concurrency?

- Assume we want to find the birth months of all laureates between 2000 and 2022
- We use something similar to the previous example
- One call per field and year, and then another per laurate
  - about 400 calls the API
- Can take minutes, most of it waiting for I/O

Creating a server

A server implementation in Go

```go
func getTime(w http.ResponseWriter, r *http.Request) {
    ct := time.Now()
    io.WriteString(w, ct.Format(time.RFC1123))
}

func main() {
    http.HandleFunc("/time", getTime)
    http.ListenAndServe(":3000", nil)
}
```

And a client in Python

```python
import requests

r = requests.get('http://localhost:3000/time')
if r.ok:
    print(f'The time is {r.text}')
```

Latency

- As discussed, there is some latency in calling the API
- Same machine, about 0.002 seconds per call
- Same network, about 0.014 seconds per call

Latency

- We can create a server that simulates latency
- Latency from, e.g.,
  - computation
  - networks
  - databases and other server I/O
  - server load
  - ...

Latency

```go
func simLatency(w http.ResponseWriter, r *http.Request) {
  time.Sleep(2*time.Second)
  io.WriteString(w, "ok")
}

// ...

http.HandleFunc("/latency", simLatency)
```

No surprise

- » Unsurprisingly, 100 calls to the API take about 200 seconds
- » If we use ten threads to make the calls, it takes about 20 seconds
    - » Surprising?

Threaded client

```python
def make_req(n):
    for i in range(n):
        r = requests.get('http://localhost:3000/latency')
        if not r.ok:
            print("req failed...", r.status_code)

ts = [Thread(target=make_req, args=(10,)) for i in range(10)]
for t in ts:
    t.start()
for t in ts:
    t.join()
```

What did we learn?

- Remember the Python GIL?
  - no problem, since threads are mostly blocked waiting for I/O
- Our server can handle ten concurrent requests
  - the handler functions run as goroutines…
  - and we mainly sleep while serving the request
  - which is I/O wait

Can we do 100?

- Yes, 100 threads take a little more than two seconds
- If we fake work instead of sleeping? $100\,000^2$ multiplications per call
- About 300 seconds for serial, 30 for ten threads, and 6 for 100 threads
  - Remember, server compute is not infinite
  - So, do not hammer servers (more later)

Creating an API

A simple API to calculate the area

- Pass height and width to an endpoint
- Compute the area and return the value
- Encoded as JSON

Structs and tags

```go
type Dimensions struct {
    Height int `json:"height"`
    Width int `json:"width"`
}

type Area struct {
    Area int `json:"area"`
}
```

Checking for the method

```go
func CalcArea(w http.ResponseWriter, r *http.Request) {
    switch r.Method {
    case "POST":
    w.WriteHeader(http.StatusOK)
  case "GET":
    w.WriteHeader(http.StatusOK)
  default:
        w.WriteHeader(http.StatusNotFound)
    }
}
```

Handling a post

- » Find JSON in the body
- » “Decode” to a struct
- » Compute the area
- » “Encode” the area struct
- » Write the response

Handling a post

```go
case "POST":
  var dims Dimensions
  data, _ := io.ReadAll(r.Body)

  if err := json.Unmarshal(data, &dims); err == nil {
    area := Area{dims.Height * dims.Width}
    if res, err := json.Marshal(area); err == nil {
      w.Write(res)
    }
  }
```

BAD!

- Works if everything is “perfect”
- Cannot assume that!
- So, not enough error handling!

Client for post

```python
import requests
import json

vals = {'height':10, 'width':20}
r = requests.post('http://localhost:3000/area', json=vals)
if r.ok:
    jo = r.json()
    print(f'The area is {jo["area"]}')
```

Handling a get

- Extract the query params from the URL
  - `...?a=b&c=d&...`
- … (same as post)

Handling a get

```
case "GET":
  height, _ := strconv.Atoi(r.URL.Query().Get("height"))
  width, _ := strconv.Atoi(r.URL.Query().Get("width"))
  area := Area{height * width}

  if res, err := json.Marshal(area); err == nil {
      w.Write(res)
  }
```

Client for get

```python
import requests
import json

vals = {'height':10, 'width':20}
url = f'http://localhost:3000/area?height={vals["height"]}&width={vals["width"]}'
r = requests.get(url)
if r.ok:
    jo = r.json()
    print(f'The area is {jo["area"]}')
```

Client for get

```python
import requests
import json

vals = {'height':10, 'width':20}
r = requests.get('http://localhost:3000/area', data=vals)
if r.ok:
    jo = r.json()
    print(f'The area is {jo["area"]}')
```