Principles of Concurrency

Lecture 16
Erlang and Go
Suppose we have N threads (or processes, tasks) that form a ring. Each thread communicates with its neighbor forwarding a message. How big can we make the ring? How long does it take to send a message?
Erlang Philosophy

Independent processes
  • suitable for executing on distributed machines

No sharing
  • (Deep) copy data sent on messages
    no cross-machine pointers
    no locks, data races, synchronization issues, ...

All processes have a unique name

Asynchronous sends, synchronous receives
  • Eventual delivery
  • But if A sends messages m1 and m2 to B, m2 will never arrive before m1
  • guarded commands
  • dynamic thread creation
  • synchronous message passing
    an input action in a guarded command causes the actions in the guard to block until the input action can be satisfied
    - there is a matching output action
  • No communication through global (shared) variables
Key features

Functional
  • single assignment (every variable assigned to at most once)

Lightweight first-class processes

Pattern-matching

Small collection of datatypes
  • lists, tuples, pairs

Dynamic typing

Realtime concurrent garbage collection
Examples

-module(math).
-export([fac/1]).

fac(N) when N > 0 -> N * fac(N-1);
fac(0) -> 1.

> math:fac(25).
15511210043330985984000000

lookup(Key, {Key, Val, _, _}) -> {ok, Val};
lookup(Key, {Key1, Val, S, B}) when Key<Key1 -> lookup(Key, S);
lookup(Key, {Key1, Val, S, B}) -> lookup(Key, B);
lookup(Key, nil) -> not_found.
Examples

append([H|T], L) -> [H|append(T, L)];
append([], L) -> L.

sort([Pivot|T]) ->
    sort([X|X <- T, X < Pivot]) ++
    [Pivot] ++
    sort([X|X <- T, X >= Pivot]);
sort([]) -> [].

> Adder = fun(N) -> fun(X) -> X + N end end.
#Fun
> G = Adder(10).
#Fun
> G(5).
15
-module(m).
-export([loop/0]).

loop() ->
    receive
        who_are_you ->
            io:format("I am ~p~n", [self()]),
            loop()
    end.

1> P = spawn(m, loop, []).
<0.58.0>
2> P ! who_are_you.
I am <0.58.0>
who_are_you
-module(counter).
-export([start/0, loop/1]).

start() ->
    spawn(counter, loop, [0]).

loop(Val) ->
    receive
        increment -> loop(Val + 1)
    end.

Issues:
- Cannot directly access counter value.
- Messaging protocol is explicit (via message increment)
-module(counter).
-export([start/0,loop/1,increment/1,value/1,stop/1]).

%% First the interface functions.
start() ->
  spawn(counter, loop, [0]).

increment(Counter) -> Counter ! increment.

value(Counter) ->
  Counter ! {self(),value},
  receive
    {Counter,Value} -> Value
  end.

stop(Counter) -> Counter ! stop.

loop(Val) ->
  receive
    increment -> loop(Val + 1);
    {From,value} -> From ! {self(),Val}, loop(Val);
    stop -> true;
    Other -> loop(Val)
  end.
-module(m).
-export([start/0, ping/1, pong/0]).

ping(0) ->
    pong ! finished,
    io:format("ping finished~n", []);
ping(N) ->
    pong ! {ping, self()},
    receive pong ->
        io:format("Ping received pong~n", [])
    end,
    ping(N - 1).

pong() ->
    receive
        finished -> io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() -> register(pong, spawn(m, pong, [])),
            spawn(m, ping, [3]).
Example

module(prodcon).
-export([start/0, consumer/0, producer/3]).

producer(_, _, 0) -> true;
producer(Me, Server, N) ->
   Server ! {Me, N},
   producer(Me, Server, N-1).

consumer() ->
   receive
      {Them, N} ->
         io:format("~s ~w~n", [Them, N]),
         consumer()
   end.

start() ->
   Server = spawn(prodcon, consumer, []),
   spawn(prodcon, producer, ['A', Server, 10]),
   spawn(prodcon, producer, ['B', Server, 5]),
   io:format("finished start~n", []).

Asynchronous, but no guarantees or notifications to the producer that the consumer has actually received its messages.
-module(prodcon).
-export([start/0, consumer/0, producer/3]).

producer(_, _, 0) -> true;
producer(Me, Server, N) ->
    Server ! {self(), Me, N},
    receive {Server, ok} ->
        true
    end,
    producer(Me, Server, N-1).

consumer() ->
    receive
        {Pid, Them, N} ->
            io:format("~s ~w~n", [Them, N]),
            Pid ! {self(), ok},
            consumer()
    end.

start() ->
    Server = spawn(prodcon, consumer, []),
    spawn(prodcon, producer, ['A', Server, 10]),
    spawn(prodcon, producer, ['B', Server, 5]),
    io:format("finished start~n", []).
Can generalize previous example to a distributed environment

-module(m).
-export([[start/0, ping/2, pong/0]]).

ping(0,Pong_Node) ->
    {pong, Pong_Node} ! finished,
    io:format("ping finished-n", []).
ping(N,Pong_Node) ->
    {pong, Pong_Node} ! {ping, self()},
    receive pong ->
        io:format("Ping received pong-n", []),
        end,
    ping(N - 1, Pong_Node).

pong() ->
    receive
        finished -> io:format("Pong finished-n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping-n", []),
            Ping_PID ! pong,
            pong()
        end.

start_pong() -> register(pong, spawn(m, pong, [])),
    start_ping(Pong_Node) -> spawn(m, ping, [3, Pong_Node]).

On one host: erl -sname ping
On another: erl -sname pong

On one node:
    M:start_pong().
On another:
    M:start_ping(pong@<host>).
...  
process_flag(trap_exit, true),
Pid = spawn_link(fun() -> ... end),
receive
    {'EXIT', Pid, Why} ->
        ...
    ...
end
server(Fun, Data) ->
    receive
        {new_fun, Fun1} ->
            server(Fun1, Data);
        {rpc, From, ReplyAs, Q} ->
            {Reply, Data1} = Fun(Q, Data),
            From ! {ReplyAs, Reply},
            server(Fun, Data1)
    end.

rpc(A, B) ->
    Tag = new_ref(),
    A ! {rpc, self(), Tag, B},
    receive
        {Tag, Val} -> Val
    end
Concurrency Patterns

**Unicast**

```plaintext
A ! B
```

**Event Handling**

```plaintext
receive A -> A end
```

**RPC call**

```plaintext
Call (RPC)
A ! {self(), B},
receive
{A, Reply} ->
Reply
end
```

**Callback**

```plaintext
receive
{From, A} ->
From ! F(A)
end
```
Callback within RPC

A ! {Tag, X}, g(A, Tag).

g(A, Tag) ->
receive
  {Tag, Val} -> Val;
  {A, X} ->
    A ! F(X),
    go(A, Tag)
end.
Timeouts

receive
  Message1 [when Actions1 ;
  Message2 [when Actions2 ;
...

after
  TimeOutExpr ->
  ActionsT
end

get_event() ->
  receive
    {mouse, click} ->
      receive
        {mouse, click} ->
          double_click
          after double_click_interval() -> single_click
      end ...
  end. 
- Developed in 2007 at Google
  - open source
- Key features:
  - compiled, statically-typed
  - garbage collected
  - C-like syntax
  - built-in concurrency
    - encourages message-passing
Concurrency

- Goroutine:
  - a function that executes concurrently with other goroutines in the same address space
  - acts as a lightweight user-space thread

- Channel:
  - generalization of Unix pipes
  - similar to channels in CSP

- Coordination primarily via channels; mutexes, locks, semaphores much less common

- Unlike Erlang, Go assumes shared address space
A goroutine is a lightweight thread of execution.

```go
package main

import (
    "fmt"
    "time"
)

func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
    }
}

func main() {
    f("direct")

    go f("goroutine")

    go func(msg string) {
        fmt.Println(msg)
    }("going")

    time.Sleep(time.Second)
    fmt.Println("done")
}
```

Establishes an asynchronous thread of control

$ go run goroutines.go
direct : 0
direct : 1
direct : 2
goroutine : 0
going
goroutine : 1
goroutine : 2
done
Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine.

```go
package main

import "fmt"

func main() {
    messages := make(chan string)

    go func() { messages <- "ping" }()

    msg := <-messages
    fmt.Println(msg)
}
```

By default, sends and receives block until both the sender and receiver are ready. This property allowed us to wait at the end of our program for the "ping" message without having to use any other synchronization.

Next example: Channel Buffering.

by Mark McGranaghan and Eli Bendersky | source | license
Buffering

By default channels are unbuffered, meaning that they will only accept sends (channel <-) if there is a corresponding receive (<- channel) ready to receive the sent value.

Buffered channels accept a limited number of values without a corresponding receiver for those values.

```go
package main

import "fmt"

func main() {
    messages := make(chan string, 2)
    messages <- "buffered"
    messages <- "channel"

    fmt.Println(<-messages)
    fmt.Println(<-messages)
}
```

Allows sends to be asynchronous
We can use channels to synchronize execution across goroutines. Here’s an example of using a blocking receive to wait for a goroutine to finish. When waiting for multiple goroutines to finish, you may prefer to use a `WaitGroup`.

```go
package main

import (
    "fmt"
    "time"
)

func worker(done chan bool) {
    fmt.Println("working..."))
    time.Sleep(time.Second)
    fmt.Println("done")
    done <- true
}

func main() {
    done := make(chan bool, 1)
    go worker(done)

    <-done
}
```

Blocks until acknowledgement received from worker.
Directionality

When using channels as function parameters, you can specify if a channel is meant to only send or receive values. This specificity increases the type-safety of the program.

```go
package main

import "fmt"

func ping(pings chan<- string, msg string) {
    pings <- msg
}

func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}

func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pons)
}
```

Channel actions (sends, receives, or both) recorded as part of its type.

Ping can only send messages on channel `pings`

Pong can only receive messages on channel `pings` and can only send messages on channel `pongs`
Select

package main

import (
    "fmt"
    "time"
)

func main() {
    c1 := make(chan string)
    c2 := make(chan string)

    go func() {
        time.Sleep(1 * time.Second)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(2 * time.Second)
        c2 <- "two"
    }()

    for i := 0; i < 2; i++ {
        select {
            case msg1 := <-c1:
                fmt.Println("received", msg1)
            case msg2 := <-c2:
                fmt.Println("received", msg2)
        }
    }
}

$ time go run select.go
received one
received two
real  0m2.245s

Sleeps on both goroutines execute concurrently
Example

- Web crawler
  - mask latency of network communication
  - access pages in parallel
  - send requests asynchronously
    - display results on receipt

```go
func main() {
    start := time.Now()
    for _, site := range os.Args[1:] {
        count("http://" + site)
    }
    fmt.Printf("%.2fs total\n", time.Since(start).Seconds())
}

func count(url string) {
    start := time.Now()
    r, err := http.Get(url)
    if err != nil {
        fmt.Printf("%s: %s\n", url, err)
        return
    }
    n, _ := ioutil.Copy(ioutil.Discard, r.Body)
    r.Body.Close()
    dt := time.Since(start).Seconds()
    fmt.Printf("%s %d [%.2fs]\n", url, n, dt)
}
```

No parallelism
Example using goroutines

```go
func main() {
    start := time.Now()
    for _, site := range os.Args[1:] {
        count("http://" + site)
    }
    fmt.Printf("%.2fs total\n", time.Since(start).Seconds())
}

func count(url string) {
    start := time.Now()
    r, err := http.Get(url)
    if err != nil {
        fmt.Printf("%s: %s\n", url, err)
        return
    }
    n, _ := io.Copy(ioutil.Discard, r.Body)
    r.Body.Close()
    dt := time.Since(start).Seconds()
    fmt.Printf("%s %d [%s%.2fs]\n", url, n, dt)
}
```