Principles of Concurrency

Week 9

Message Passing

Issues

Given a collection of threads, how should they communicate information among one another?

Using message-passing, they communicate through *messages*, information that is directed from one thread to another.

- Sometimes the recipient may be anonymous
  - channel-based communication
- Typically, the recipient is known

What should the sender do after the message is sent?

- wait until the recipient acknowledges receipt (synchronous)
- proceed regardless (asynchronous)

Communicating Sequential Processes

Landmark proposal by Hoare in 1978

Key components

- 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*

Parallel Commands

Defines eleven processes

Behavior of processes fork(0), ..., fork(4) specified by command FORK

- Bound variable i indicates identity of a particular fork

Similar structure for phil

```
room :: ROOM ||
fork(i:0..4) :: FORK ||
phil(i:0..4) :: PHIL
```

Input and Output Commands

`X ? (a,b)`

- inputs from process X a pair, binding the first element to a and the second to b

`Y ! (3 * a, b + 13)`

- outputs to process Y a pair, consisting of the values computed by the corresponding expressions within the environment in which the command takes place

`X(i)? V()`

- From the ith array of processes X, input a signal V()

`display(i-2) ! “A”`

- send to the i-2nd display the character “A”

Guarded Commands


```
x >= y --> m := x  []  y >=x m := y
```

Assign m to x if x >= y; assign m to y if y >= x. Do one or the other if x = y.

```
i := 0;
* [ i < size; content(i) <> n -> i := i + 1 ]
```

Scan the elements of the array contents incrementing counter i as long as n is not encountered and the end of the array is not reached.

Guarded Commands


```
X:: *[c:char, A?c ->
         [ c <> “*” --> B!c
         []
           c = “*” --> A?c;
                          [ c <> “*” --> B!”*”; B!c
                          []
                            c = “*” --> B!”#” ]
         ]  ]
```

What does this program do?

What assumptions does it make?

Bounded Buffer


```
X::
  buffer:(0..9) portion;
  in,out:integer, in:= 0; out := 0;
  *[in < out + 10; producer?buffer(in mod 10) --> in := in + 1
   []
    out < in; consumer?more() --> consumer ! buffer(out mod 10);
                                  out := out + 1
  ]
```

Consumer contains pairs of commands `X!more()` and `X?p`

Producer contains commands of the form `X!p`

Small Set of Integers


```
S::
  content:(0..99)integer, size:integer,size := 0;
*[  n:integer,X?has(n) --> SEARCH;X!(i<size)
    []
    n:integer; X?insert(n) --> SEARCH;
       [ i<size --> skip
         []
         i = size; size < 100 -->  content(size) := n; size := size + 1
]   ]
```

where SEARCH is:

```
i:integer; i := 0;
 *[i < size; content(i) <> n --> i := i + 1 ]
```

Dining Philosophers

Five philosophers:

- Only eat and think
- Share a common dining room.
  - Shared bowl of spaghetti
  - Five forks
- Need two forks to eat (both right and left)
- After finishing eating, puts both forks down

Dining Philosophers


```
PHIL = *[ ... for ith philosopher ....
               THINK;
                room!enter( );
                fork(i)!pickup(); fork((i+1) mod 5)!pickup();
               EAT;
                 fork(i)!putdown(); fork((i+1) mod 5)!putdown();
                 room!exit()
              ]

FORK = *[phil(i)?pickup() --> phil(i)?putdown()
               | (phil(i - 1) mod 5)?pickup() --> phil((i-1) mod 5)?putdown()
              ]

ROOM = occupancy:integer; occupancy := 0;
               *[(i:0..4)phil(i)?enter() --> occupancy := occupancy + 1
               | (i:0..4)phil(i)?exit() --> occupancy := occupancy - 1
              ]

[room::ROOM || fork(i:0..4)::FORK || phil(i:0..4)::PHIL]
```

What happens if all five philosophers enter the room, and each picks up the left fork?
How would you adapt the algorithm to prevent this scenario?

Issues

Explicit naming of source and destination

- No first-class channels or ports

Fully synchronous

- How would you model asynchronous communication?

No unbounded number of processes

Fairness

```
[X::Y!stop() || Y::continue:boolean; continue := true;
                *[ continue; X?stop( ) --> continue := false
                 |  continue --> n := n + 1
               ]
]
```

Output guards

`Z:: [X!2 || Y!3]` could be expressed as: `Z::[X!2 --> Y!3 [] Y!3 --> X!2]`

Why does the following not work?

```
Z::[true --> X!2; Y!3 [] true --> Y!3; X!2]
```

Consider: `Y :: Z?y; X!go() || X:: Y?go(); Z?x`

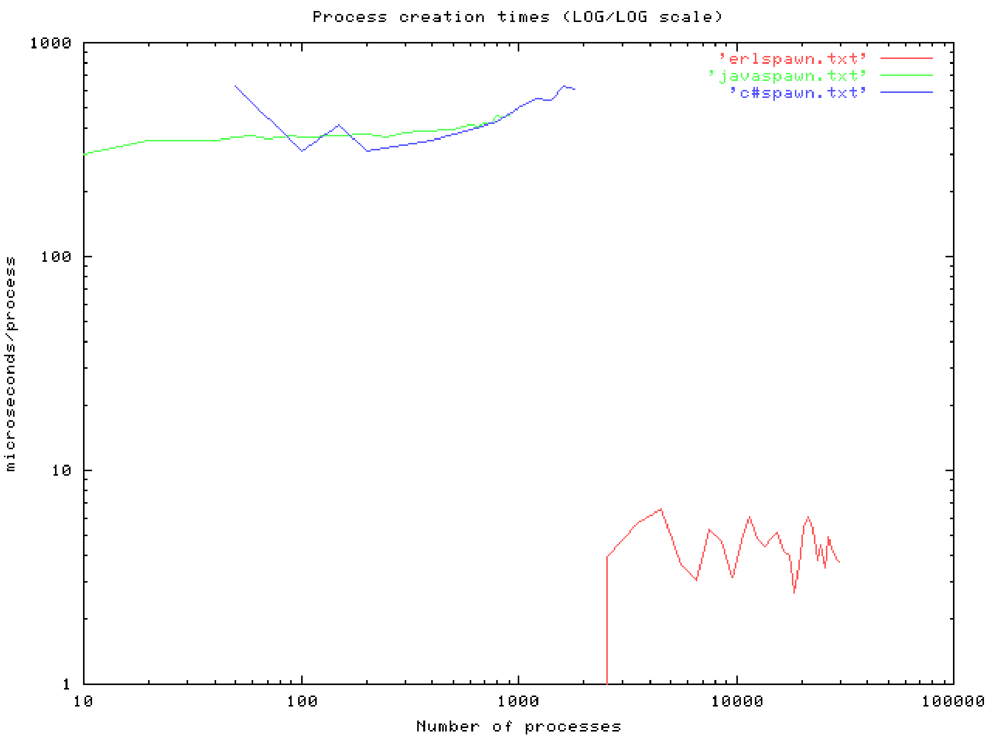
Example

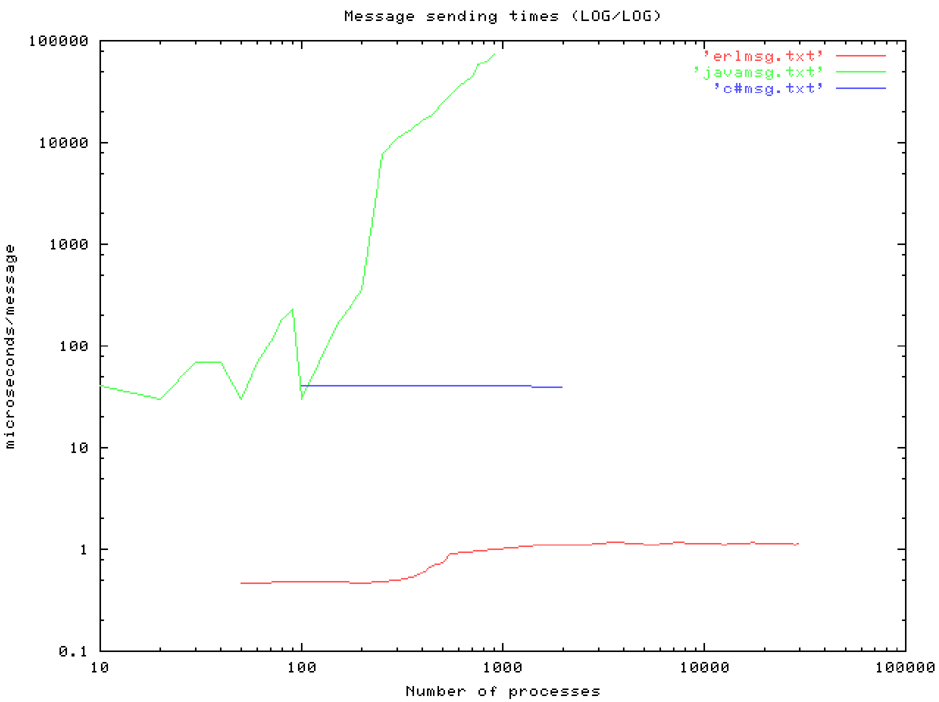
Suppose we have N threads (or processes, tasks) that form a ring

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
```

Concurrency


```
-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
```

Concurrency


```
-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)

Refinement


```
-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.
```

Concurrency


```
-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.

Acks


```
-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", []).
```

Distributed Programming

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>).
```

Monitoring


```
...
process_flag(trap_exit, true),
Pid = spawn_link(fun() -> ... end),
receive
   {'EXIT', Pid, Why} ->
       ...
end
```

Client/Server


```
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

```
A ! B
```

RPC call

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

Event Handling

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

Callback

```
receive
  {From, A} ->
    From ! F(A)
end
```

Concurrency Patterns

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.
```

Go

- 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

Goroutine


```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


```
package main

import "fmt"

func main() {

    messages := make(chan string)


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



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

sends a message onto a channel

receives a message from a channel

Buffering


```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

Channel Synchronization


```go
package main

import (
    "fmt"
    "time"
)

func worker(done chan bool) {
    fmt.Print("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


```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(<-pongs)
}
```

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

Ping can only send messages on chan

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

Select


```go
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

```
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 [%.2fs]\n", url, n, dt)
}
```

No parallelism

Example using goroutines


```
func main() {
  start := time.Now()
  c := make(chan string)
  n := 0
  for _, site := range os.Args[1:] {
    n++
    go count("http://" + site, c)
  }
  for i := 0; i < n; i++ {
    fmt.Print(<-c)
  }
  fmt.Printf("%.2fs total\n", time.Since(start).Seconds())
}
func count(url string, c chan<- string) {
  start := time.Now()
  r, err := http.Get(url)
  if err != nil {
    c <- fmt.Sprintf("%s: %s\n", url, err)
    return
  }
  n, _ := io.Copy(ioutil.Discard, r.Body)
  r.Body.Close()
  dt := time.Since(start).Seconds()
  c <- fmt.Sprintf("%s %d [%.2fs]\n", url, n, dt)
}
```