Concurrent Programming: Languages and Techniques

Lecture 1: Introduction

15 September 2022

MIEI - Integrated Masters in Comp. Science and Informatics
Specialization Block

Bernardo Toninho
(with António Ravara and Carla Ferreira)

NOVALINCS

Part I
Administrivia

Administrivia

3 Modules:

1. Message-passing Concurrency (Go)
2. Message-passing Conc. + Distribution (Erlang)
3. Memory safe Shared-memory Concurrency (Rust)

Administrivia

Grading:

- 3 Mini-Projects (**5%** each, 2 best)
- 3 Projects (**20%** each)
- 3 Mini-tests (**10%** each, no min. grade)

Final grade is **70%** projects (and mini) + **30%** tests

Administrivia

Projects and Mini-Projects:

- Groups of 2 students (3 only if approved!)
- Mini-projects are 1 week-long small scale
- Projects are 2 week-long medium/small scale

Administrivia

Mini-tests (~1h30 duration):

- After each module, covers material from the module.
- Multiple choice (12 points), focusing more on the concepts / theoretical aspects.
- Open answer (8 points)
- Includes (open answer) questions about the project!
- Sample test will be made available in due time.

Administrivia

Typical module structure:

- Week 1: Intro to language & prog. paradigm
- Week 2: Mini-Project handout (due following week)
- Week 3: Project handout (due in 2 weeks)
- Week 4: Final lecture and project support

Administrivia

Important dates (tentative):

- Week of Oct-3: Mini-Project 1 Deadline
- Week of Oct-17: Project 1 Deadline
- Week of Oct-24: **Mini-Test 1**
- Week of Oct-31: Mini-Project 2 Deadline
- Week of Nov-14: Project 2 Deadline
- Week of Nov-21: **Mini-Test 2**
- Week of Nov-28: Mini-Project 3 Deadline
- Week of Dec-12: Project 3 Deadline
- Week of Dec-19: **Mini-Test 3**

Part II
Language-based Problem Solving

Programming is Hard!

- Programming is about convincing machines to do what we want them to do.
- Systematic and rigorous (“No more, no less”).
- The way in which we communicate with machines shapes what we can (or can’t) do.
- …and how easy it is achieve it.

Programming is Hard!

- Programming languages and their features shape what our programs can and cannot do.
- “If the only tool you have is a hammer, you tend to see every problem as a nail.”
- Lets think a bit about what kind of “hammers” we have…

What about PLs?

- Two axes:
  - What kind of runtime errors are allowed?
  - What idioms and paradigms are facilitated by language features?

Things go Wrong

- Different languages allow programs to go wrong in different ways.
- We distinguish between runtime and compile-time errors:
  - Anything goes (e.g. Python)
  - Memory violation errors (e.g. C/C++)
  - Null pointer dereferencing (e.g. Java)
  - Memory safety (e.g. Rust)

Languages as Tools

- Beyond errors, languages also provide abstractions that can be better suited for certain problems.
- Languages and their implementations have a very wide range of focus.
- Different problems are better solved using specific (language) features.

Languages as Tools

Writing a compiler / interpreter / static analyzer?

```haskell
data Type
    = Base
    | Arrow Type Type
    deriving (Eq, Ord, Read, Show)

data Term
    = Const
    | Var Int -- deBruijn indexing; the nearest enclosing lambda binds Var 0
    | Lam Type Term
    | App Term Term
    deriving (Eq, Ord, Read, Show)

check :: [Type] -> Term -> Maybe Type
check env Const = return Base
check env (Var v) = atMay env v
check env (Lam ty tm) = Arrow ty <$> check (ty:env) tm
check env (App tm tm') = do
    Arrow i o <- check env tm
    i' <- check env tm'
    guard (i == i')
    return o

eval :: Term -> Term
eval (App tm tm') = case eval tm of
    Lam _ body -> eval (subst 0 tm' body)
eval v = v

subst :: Int -> Term -> Term -> Term
subst n tm Const = Const
subst n tm (Var m) = case compare m n of
    LT -> Var m
```

VS

- Expr.java
- ExprBetaReducer.java
- ExprBuilder.java
- ExprEtaReducer.java
- ExprNone.java
- ExprParser.java
- ExprParserUntyped.java
- ExprPredicate.java
- ExprPrinter.java
- ExprRichBuilder.java
- ExprToDeBruijn.java
- ExprToFreeNames.java
- ExprToType.java

Languages as Tools

Web development?

```html
<html>
  <head>
    <div>
        <div>

            <form method="post" action="#" id="formvalue" onkeyup="
            drawChart()" />

            </form>
        </div>
    </div>



    <script type="text/javascript" src="https://www.google.com/jsapi"></
      script>
    <script type="text/javascript">

    var bid = 43;
    var ask = 21;


      google.load("visualization", "1", {packages:["corechart"]});
      google.setOnLoadCallback(drawChart);
      function drawChart() {
        var data = google.visualization.arrayToDataTable([
          ['Price', 'Quantity'],
          ['Value #1', bid],
          ['Value #2', ask],
        ]);
```

Javascript

```typescript
type filterFn = (item: string) => bool;

// The higher-order-function takes an array and a function as arguments
function filterItems(arr: string[], fn: filterFn): string[] {
    const newArray: string[] = [];
    arr.forEach(item => {
        if(fn(item)){
          newArray.push(item);
        }
    });
    return newArray;
}

function checkNameLength(name: string) {
  return name.length >= 10;
}

const doctorList = ["DoctorOne", "DoctorTwo", "DoctorThree", "DoctorFour"];

// We are passing the array and a function as arguments to filterItems method.
const output = filterItems(doctorList, checkNameLength);

console.log(output); // ["DoctorThree", "DoctorFour"]
```

Typescript

```elm
view : Model -> Html Msg
view model =
    div
        [ class "todomvc-wrapper"
        , style [ ( "visibility", "hidden" ) ]
        ]
        [ section
            [ class "todoapp" ]
            [ lazy viewInput model.field
            , lazy2 viewEntries model.visibility model.entries
            , lazy2 viewControls model.visibility model.entries
            ]
        , infoFooter
        ]


viewInput : String -> Html Msg
viewInput task =
    header
        [ class "header" ]
        [ h1 [] [ text "todos" ]
        , input
            [ class "new-todo"
            , placeholder "What needs to be done?"
            , autofocus True
            , value task
            , name "newTodo"
            , onInput UpdateField
            , onEnter Add
            ]
            []
        ]
```

Elm

Languages as Tools

Device driver?

```c
#include <linux/module.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

// module attributes
MODULE_LICENSE("GPL"); // this avoids kernel taint warning
MODULE_DESCRIPTION("Device Driver Demo");
MODULE_AUTHOR("Appu Sajeev");

static char msg[100]={0};
static short readPos=0;
static int times = 0;

// protoypes,else the structure initialization that follows fail
static int dev_open(struct inode *, struct file *);
static int dev_rls(struct inode *, struct file *);
static ssize_t dev_read(struct file *, char *, size_t, loff_t *);
static ssize_t dev_write(struct file *, const char *, size_t, loff_t *);

// structure containing callbacks
static struct file_operations fops =
{
        .read = dev_read, // address of dev_read
        .open = dev_open,  // address of dev_open
        .write = dev_write, // address of dev_write
        .release = dev_rls, // address of dev_rls
};


// called when module is loaded, similar to main()
int init_module(void)
{
        int t = register_chrdev(89,"myDev",&fops); //register driver with major:89

        if (t<0) printk(KERN_ALERT "Device registration failed..\n");
        else printk(KERN_ALERT "Device registered...\n");

        return t;
}


// called when module is unloaded, similar to destructor in OOP
void cleanup_module(void)
{
```

C/C++

```rust
/// Communicating to the sensor, updates the latest wrench.
/// # Returns
/// `Ok(wrench)` if succeeds, `Err(reason)` if failed.
pub fn update(&mut self) -> Result<Wrench, Error> {
    let res = receive_message(&mut self.port);

    // Regardless of success or failure of receive_message(), request the next single data.
    // If we do not so, after updating failed once, updating will fail everytime due to no reception from the sensor.
    self.request_next_wrench()?;

    let res = match res {
        Ok(res) => res,
        Err(e) => {
            return Err(e);
        }
    };

    let (fx, fy, fz, mx, my, mz) = (0..6)
        .map(|i| 4 + i * 2)
        .filter_map(|start| res.get(start..start + 2))
        .map(|res| i16::from_le_bytes(res.try_into().unwrap()))
        .map(|digital| digital as f64)
        .next_tuple()
        .ok_or(Error::ParseData)?;

    let rated_binary = self.product.rated_binary();
    let force = Triplet::new(fx, fy, fz)
        .map_entrywise(self.rated_wrench.force, |left, right| {
            left / rated_binary * right
        });
    let torque = Triplet::new(mx, my, mz)
        .map_entrywise(self.rated_wrench.torque, |left, right| {
            left / rated_binary * right
        });

    self.last_raw_wrench = Wrench::new(force, torque);

    Ok(self.last_wrench())
}
```

Rust

Languages as Tools

Operating System?

```rust
/// An ELF executable
pub struct Elf<'a> {
    pub data: &'a [u8],
    header: &'a header::Header
}

impl<'a> Elf<'a> {
    /// Create a ELF executable from data
    pub fn from(data: &'a [u8]) -> Result<Elf<'a>, String> {
        if data.len() < header::SIZEOF_EHDR {
            Err(format!("Elf: Not enough data: {} < {}", data.len(), header::SIZEOF_EHDR))
        } else if &data[..header::SELFMAG] != header::ELFMAG {
            Err(format!("Elf: Invalid magic: {:?} != {:?}", &data[..header::SELFMAG], header::ELFMAG))
        } else if data.get(header::EI_CLASS) != Some(&header::ELFCLASS) {
            Err(format!("Elf: Invalid architecture: {:?} != {:?}", data.get(header::EI_CLASS), header::ELFCLASS))
        } else {
            Ok(Elf {
                data,
                header: unsafe { &*(data.as_ptr() as usize as *const header::Header) }
            })
        }
    }

    pub fn sections(&'a self) -> ElfSections<'a> {
        ElfSections {
            data: self.data,
            header: self.header,
            i: 0
        }
    }

    pub fn segments(&'a self) -> ElfSegments<'a> {
        ElfSegments {
            data: self.data,
            header: self.header,
            i: 0
        }
    }
```

Rust

```ocaml
module Rx(Time:Mirage_time.S)(ACK: Ack.M) = struct
  open Tcp_packet
  module StateTick = State.Make(Time)

  (* Individual received TCP segment
     TODO: this will change when IP fragments work *)
  type segment = { header: Tcp_packet.t; payload: Cstruct.t }

  let pp_segment fmt {header; payload} =
    Format.fprintf fmt
      "RX seg seq=%a acknum=%a ack=%b rst=%b syn=%b fin=%b win=%d len=%d"
      Sequence.pp header.sequence Sequence.pp header.ack_number
      header.ack header.rst header.syn header.fin
      header.window (Cstruct.length payload)

  let len seg =
    Sequence.of_int ((Cstruct.length seg.payload) +
    (if seg.header.fin then 1 else 0) +
    (if seg.header.syn then 1 else 0))

  (* Set of segments, ordered by sequence number *)
  module S = Set.Make(struct
      type t = segment
      let compare a b = (Sequence.compare a.header.sequence b.header.sequence)
    end)

  type t = {
    mutable segs: S.t;
    rx_data: (Cstruct.t list option * Sequence.t option) Lwt_mvar.t; (* User receive channel *)
    ack: ACK.t;
    tx_ack: (Sequence.t * int) Lwt_mvar.t; (* Acks of our transmitted segs *)
    wnd: Window.t;
    state: State.t;
  }
```

OCaml

Languages as Tools

- Different lang. impl. provide different trade-offs:
  - Efficiency
  - Correctness
  - Rapid prototyping
  - Ease of refactoring
  - Interoperability

Languages as Tools

Correctness is mandatory, but how hard it is to get it will often depend on the tool / language.

```
yuvraj@DeathNote:~/head-first-c-exercises$ ./find
Search Results:
------------------------------------
William: SBM GSOH likes sports,TV,dining
Josh:SJM likes sports,movie theatre
Segmentation fault (core dumped)
yuvraj@DeathNote:~/head-first-c-exercises$
```

Part II
Concurrency

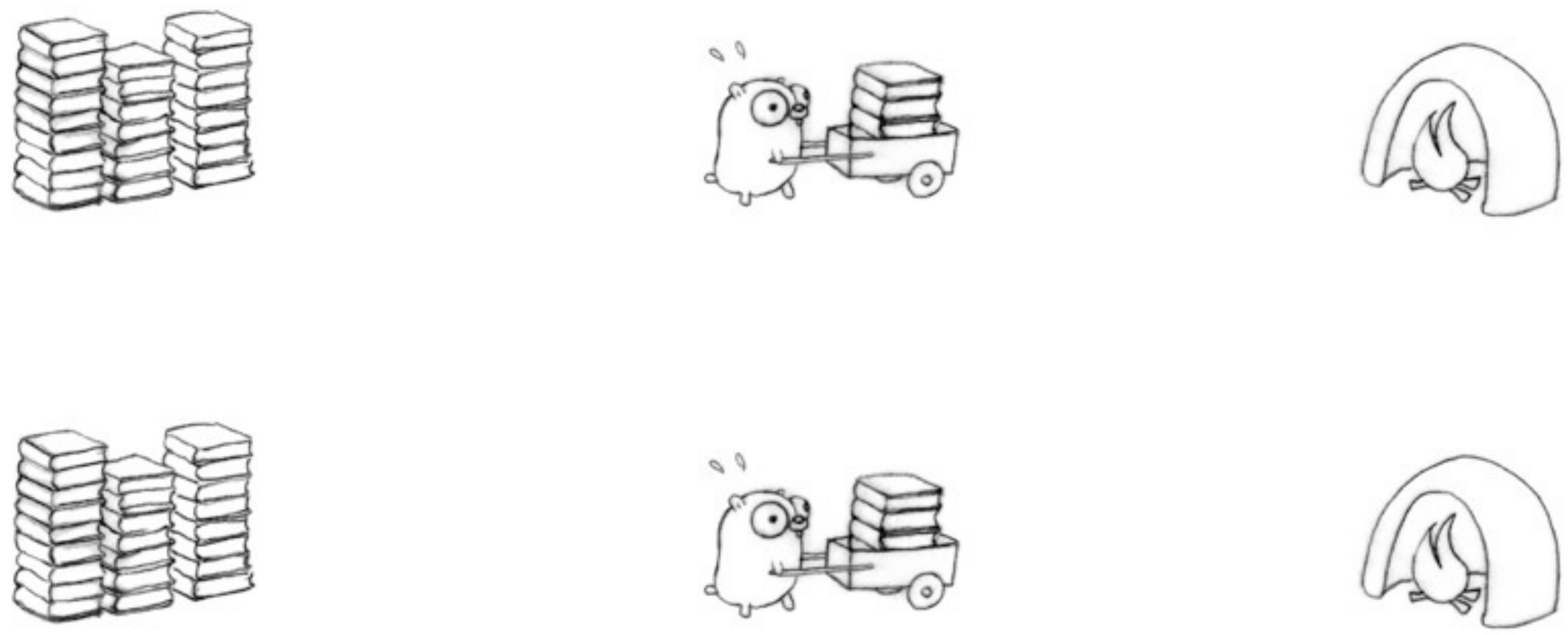
Concurrency

- In a single processor, programs executed "simultaneously"
- **Scheduler** distributes "processor time" to running programs (**processes**)
- Processes compete for processor access
- This is true **regardless** of the number of physical processors.

Parallelism vs Concurrency

Parallelism: Programming as the simultaneous execution of (possibly related) computations.

Concurrency: Programming as the composition of independently executing processes.

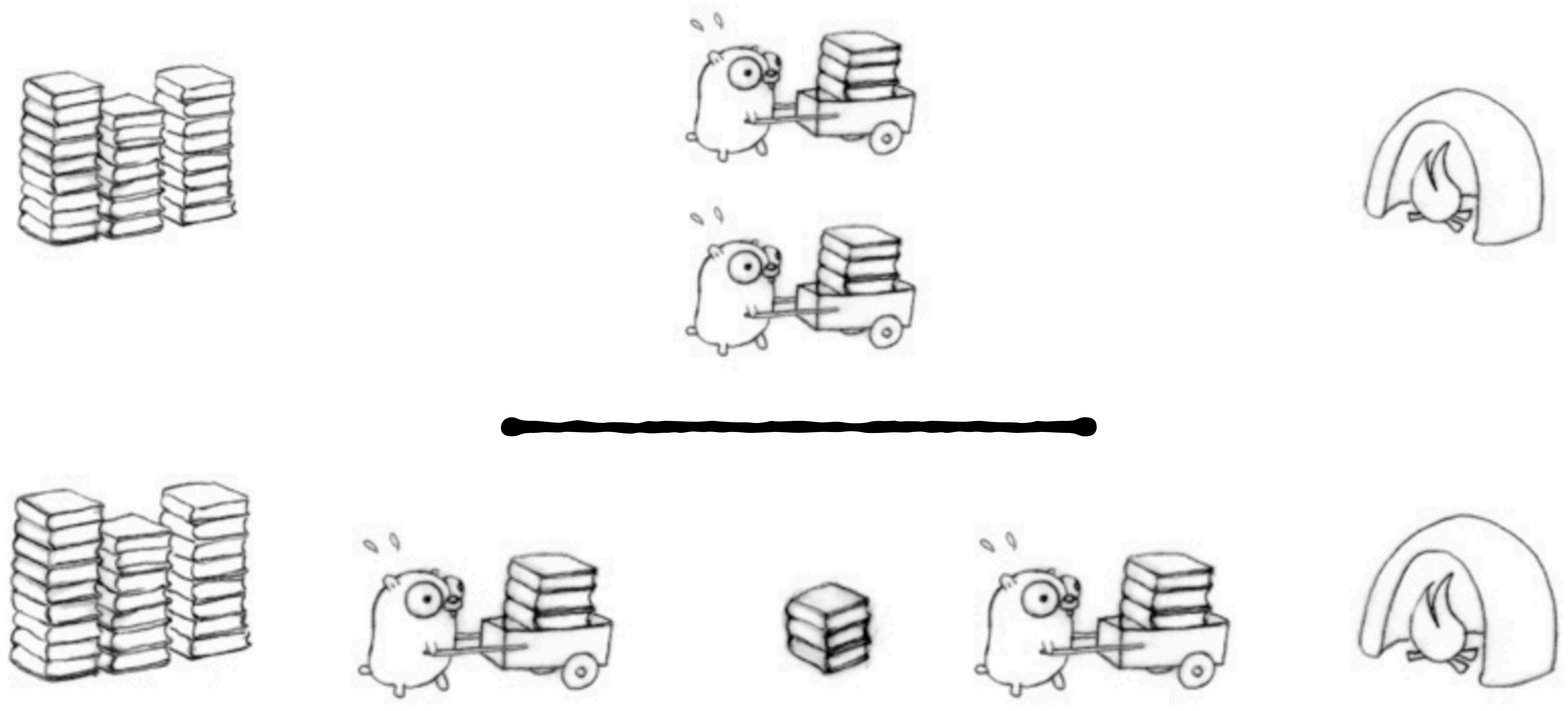


picture from https://talks.golang.org/

Parallelism vs Concurrency

Parallelism: Programming as the simultaneous execution of (possibly related) computations.

Concurrency: Programming as the composition of independently executing processes.



Concurrency vs Parallelism

- Concurrency is not parallelism, but parallelism is enabled by concurrency!
- Programs can be concurrent and have 0 parallelism.
- Well-written concurrency may run better on a multiprocessor.

Concurrency and Independence

- Concurrency is a way to **structure** work into independent pieces …

… but then you have to coordinate those pieces

Andrew Gerrand (Golang)

- "*Independent*" here refers to a way of *thinking* about problems, and *structuring* their solutions.
- Concurrent processes may indeed *interfere/interact*

Type of Concurrency

Shared memory concurrency:

- Processes coordinate by reading and writing to memory locations that are shared.

- Concurrent memory accesses managed by locks.

Message-passing concurrency:

- Processes coordinate by sending and receiving messages along **channels**.

- More abstract / higher-level than shared memory concurrency

- Channel operations need not be managed (processes may concurrently perform channel ops **safely**).

Challenges

- Reasoning about possible executions is a combinatoric problem — *state explosion problem*.

- With $P = P_1 ; P_2$ and $Q = Q_1 ; Q_2$, P and Q in parallel have 6 possible executions / interleavings.

- Errors can be quite difficult to diagnose / reproduce.

Safe Concurrency

What do we mean by safe? What can go wrong?

- Uncontrolled concurrent accesses to data **(Data Races)**

- Being stuck forever acquiring a lock **(Deadlock)**

- Being stuck forever trying to read / write to a channel **(Deadlock)**

- Repeating the same interaction without doing any useful work **(Livelock)**

Safe Concurrency

Correctness properties:

- **Mutual exclusion** (no conflicts — ensures safety/ consistency)
- **Deadlock-freedom** (the system as a whole is never fully “stuck”)
- **Progress** (No subsystem is stuck waiting forever)

Part III
Concurrency and Programming Languages

Concurrency and PLs

- Concurrency is about how processes/threads coordinate to achieve a goal.

- …but we don't program in "concurrency", we do it in a concrete PL.

- Concurrency primitives and programming paradigms change drastically from language to language.

Concurrency and PLs

Can / how do languages help in achieving correctness / ruling out errors?

- **Rust:** Resource ownership (compile-time analysis)

- **Go:** High-level concurrency realized via channels and "goroutines".

- **Erlang:** High-level concurrency realized with the actor model.

Concurrency and PLs

Rust:

- No garbage collector (a la C/C++).
- Memory safety (**not** a la C/C++!).
- Memory is managed through a system of **ownership** with a set of rules that the **compiler checks:**
  - Each value in Rust has an owner
  - There can be only one owner at a time
  - When the owner goes out of scope, the value is be dropped.

Concurrency and PLs

Rust:

- Ownership can be **borrowed.**

- (Im)mutable references generate (im)mutable borrows.

- At any given time, you can have either one mutable reference or any number of immutable references.

- Everything else is a **compiler error**.

Concurrency and PLs

Rust:

- At any given time, you can have either one mutable reference or any number of immutable references.

- Sounds a lot like the discipline to prevent data-races in shared memory concurrency, doesn’t it? More later :)

Concurrency and PLs

Go:

- Designed by Google for systems / cloud programming.

- Implementation is garbage-collected.

- Channel-based concurrency built-in.

- (Typed) channels are first-class objects.

- Lightweight threads built-in (goroutines)

Concurrency and PLs

Go:

- Threads can read/write to channels concurrently, safely… but not all concurrency woes are solved!

- Pushes developers to synchronize threads using channels.

- Provides various primitives to use and interact with channels.

- Various concurrency prog. patterns become crucial when structuring your code… more later :)

Concurrency and PLs

Erlang:

- Aimed at large telecom applications

- Distributed, reliable, soft real-time concurrent systems

- Actor-based concurrency

- Special case of message-passing concurrency

Concurrency and PLs

Erlang:

- Actors are isolated computational units

- Communicate by exchanging asynchronous messages

- Messages are queued in a mailbox and processed sequentially.

- No assumptions on message delivery guarantees.

Concurrency and PLs

Erlang:

- Robust failure-handling mechanism.
- Actors can monitor other actors and detect termination.
- “Let it crash” philosophy
- Monitors can restart / kill monitored actors.
- …more later :)

Concurrency and PLs

Some new “hammers” for specific “nails”:

- **Rust:** Compile-time guarantees of memory safety and race-freedom for shared memory concurrency. Systems / low-level programming.

- **Go:** High-level concurrency realized by “goroutines” that share memory by communicating over channels. High-level / Cloud / Backend programming.

- **Erlang:** High-level concurrency realized with the actor model. Actors exchange messages asynchronously. Failure-handling via monitor agents and supervision trees. Distributed, reliable, soft real-time concurrent systems.

Takeaway

- Programming languages should be just like any other tool in your arsenal.
- You should be flexible enough to use the right tool for the job!
- In this course, you will explore three different concurrent programming paradigms and their common idioms.
- By the end, you will have (hopefully) more tools in your toolbox!

That's it for today…

Next week:

- Message-passing concurrency module (Go)
- Intro to Go and its message-passing concurrency features
- Some simple programming exercises in Go to get you started.