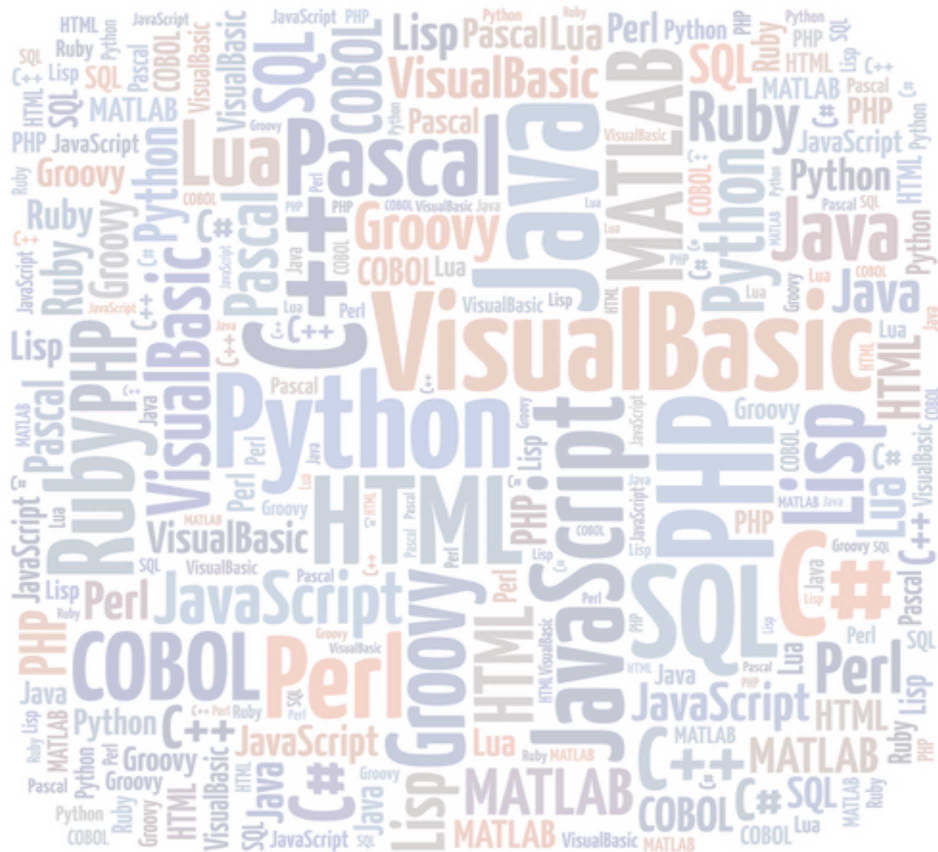
COMP 6411: Comparative Programming Languages

The Erlang Concurrency Model



Introduction

- Thread scheduling
- The Erlang way
- Spawn, send and receive
- Process identifiers
- Pattern matching on messages
- Continuous receive loops
- Receive timeouts
- Process registration

Threads and Scheduling

- Traditionally, multi-threaded code runs on top of an operating system that treats each thread as an independently scheduled entity.
- So, for example, if two applications are running on a system (database and browser), and each has two separate threads, the OS would schedule 4 threads for execution on the CPU(s).
- For modest thread counts, this works well since it allows each thread to get a full slice of the available time on the CPU.

The Erlang world view

- Erlang does not work this way.
- In the Erlang world, there may be hundreds of thousands of threads.
- It isn't possible for a typical computer to support/ schedule this many threads.
- Instead, the Erlang virtual machine creates its own super-lightweight threads that each receive a tiny portion of the time allotted to the Erlang VM process
  - The OS does not know anything about these threads.
  - This allows a massive number of threads to be created with almost no resource overhead.
  - Note, however, that all threads must share the same block of CPU time.

Erlang Message Passing

- Some message passing frameworks have hundreds of functions (i.e., the C-based Message Passing Interface or MPI).
- Erlang is MUCH simpler than this and provides just three basic primitives for creating tasks and communicating between them.
  - spawn
  - send
  - receive
- There are various ways that these functions can be used, of course, but we will look at the most basic approaches.

Process creation

- Erlang refers to its lightweight threads as processes (not to be confused with the OS concept of a "heavyweight" process)
- To create a new process, we use the `spawn` function.
- The most basic form is
  - `Pid = spawn(Mod, Func, Args)`
  - This generates a new process that begins running the Func function found in the Mod Module. A list of args can be passed to Func.
  - The spawn function returns the process ID of the new process.
  - This can be saved so that the current process has a way to contact the new process.
- IMPORTANT: There are no guarantees about whether the parent or child will execute first
  - Your application logic should not depend on a rigid send/ receive order

Trivial example

- Let's say we have the module `bar` containing the function `baz`.
- From the current module `foo`, we can create a new process to run `baz`, as depicted in the two code samples below
- Note: This example does NOT do anything useful yet since there is no communication between the two processes.

```
-module(bar).
-export(baz/2).

baz(X, Y) -> X * Y.
```

```
-module(foo).

start() ->
  PID = spawn(bar, baz, [4, 3]).
```

Communication

- Processes "talk" to one another using the `send` and `receive` functions.
- Note that processes do not share any memory so they have no direct access to the same data structure(s).
- The send function works as follows:
  - `Pid ! Message`
  - Here, Pid is the process ID of the target.
  - ! is the send operator
  - Message is one or more data items (e.g., a tuple or a list of ints).

Send and Receive…cont'd

- The `receive` is quite powerful but slightly more involved
- The idea is that we create a receive block that includes one or more patterns that can be matched against the message(s) that arrive in the process queue.
- A simple example is given below.
  - Note that the { } in the pattern indicate a tuple in this example, but you can send/receive any type of data

```
listen() ->
  receive
    {dog, Name} ->
      "Mr" ++ Name; % create a new string
    {cat, Name} ->
     "Ms" ++ Name    % create a new string
  end.
```

Receive…cont'd

- Things to keep in mind:
  - `listen` is not a special keyword. We could have called the listen function anything we wanted.
  - The receive block is terminated by `end`.
  - `receive` consists of one or more patterns
  - When the "listen" function is called, Erlang will start with the first message in the current queue.
  - It will match it against the first valid pattern that it can find. If a match is found, the code associated with the pattern is executed and the listen function terminates.
  - If no match is found, the message goes into a "save" queue and can be checked again at a later time.

Looping receives

- A receive can be invoked in two ways:
  - Periodically. Here, the point is to check for a message and then go back to doing other things
    - The previous example is like this.
  - Continuously. This would be representative of a server application (e.g., web or dbms server).
    - In this case, we would want the new process to sit in a receive loop and wait for many many messages (perhaps loop forever)
    - An example of this form is below.
    - Note how we call the listen function recursively to continue the receive process so that more than one message is processed from the queue.

```
listen() ->
  receive
    {dog, Name} ->
      dog_func(Name), % do something
      listen();
    {cat, Name} ->
      cat_func(Name),  % do something
      listen()
  end.
```

Two-way communication

- So far, we have sent a message and received a message.
- Often, we need two-way communication.
- For this, we simply need to include the sender's process ID with the message.
  - This will allow the receiver to send a message back.
- Erlang provides the `self()` function to capture the process ID of the current process

```
-module(foo).

start() ->
  PID = spawn(bar, baz, [4, 3]),
  PID ! {self(), "Important Info"}.
```

Communication…cont'd

- On the other side, we can capture the ID and return a message.
- Below, we see how the sender PID is captured and then used to send a message back
  - Note that it is perfectly valid to have a nested receive inside another receive
  - If, for example, you needed a second, related message from the sender

```
listen() ->
  receive
    {Sender, Msg} ->
      do_something(Msg),
      Sender ! "Thanks"
end.
```

Timeout options

- One problem with the receive is if the expected message never arrives (e.g., the sender crashes)
- While this is not a true deadlock, the program may hang if it waits forever for something that will never come (and the queue remains empty)
- To address this, Erlang provides a "timeout" option on the receive that can break out of the receive if the message does not arrive in a specific period of time.
  - Below, we see a timeout after 2000 milliseconds
  - Note that do_something can just be set to `true` or `ok` if no special action is required

```
listen() ->
  receive
    {Msg} ->
      do_something(Msg)
  after 2000 ->       % break after 2 seconds
    do_something()    % do this and finish
end.
```

Name registration

- So far, it has only been possible to communicate within a parent/child process pattern
  - The parent process captures the child ID
- Often, processes need to communicate with other “non-related” processes.
- Erlang provides a *name registration* service for this purpose.
- This allows a process ID to be associated with a name/label
- Other processes can then obtain the relevant ID
  - Note that the names have to be known in advance.

Registration…cont'fd

- The relevant functions are
  - `register(atomName, ID)`
  - `unregister (atomName)`
  - `whereis(atomName)` % returns PID or undefined
  - `registered( )` % returns a list of all processes

```
start() ->
  register(woof, spawn(myLib, dog)),
  register(meow, spawn(myLib, cat)).
```

```
dog() ->
  Cat = whereis(meow),
  Cat ! "dog says woof".
```

Odds and Ends

- As you can probably guess, there are more functions and features that Erlang processes can use.
  - For example, it is possible to link processes together so that "supervisor" processes can be notified if a monitored process terminates.
  - It is also possible to provide *guards* on the receive pattern so that matching is conditional
  - `receive {Flag, Name} [when Flag == 1] -> …`
- That said, the functionality presented in this guide will allow you to create and manage a wide array of concurrent Erlang processes.
  - More importantly, it will allow you to do this in a way that is inherently safe.

bar.erl, foo.erl