Processes and Messages

Mark Greenstreet and Ian M. Mitchell

CpSc 418 – January 8, 2018

- Erlang Processes
- Sending and Receiving Messages
- Best Practices with Messages
- Table of Contents

Objectives

- Introduce Erlang's features for concurrency and parallelism
  - Spawning processes.
  - Sending and receiving messages.
- Describe timing measurements for these operations and the implications for writing efficient parallel programs.
  - **Communication often dominates the runtime of parallel programs.**
- The source code for the examples in this lecture is available here: procs.erl.

Processes – Overview

- The built-in function spawn creates a new process.
- Each process has a process-id, pid.
  - The built-in function self() returns the pid of the calling process.
  - `spawn` returns the pid of the process that it creates.
  - The simplest form is `spawn(`*Fun*`)`.
    - A new process is created – "the child".
    - The pid of the new process is returned to the caller of `spawn`.
    - The function *Fun* is invoked with no arguments in that process.
    - The parent process and the child process are both running.
    - When *Fun* returns, the child process terminates. The return value is discarded.
- Operations on pids
  - send messages: `Pid ! Message`
  - debug, see: http://erlang.org/doc/apps/debugger/debugger_chapter.html and http://erlang-tutorial.blogspot.ca/2010/03/erlang-debugging.html, but I'll admit that I haven't used the debugger myself.
  - get all kinds of information about the process:

    process_info`(Pid, What)`.

Processes – a friendly example

- Writing the code:

```
hello(N) when is_integer(N), N >= 0 ->
  [    spawn(fun() -> io:format(
         "hello world from process ~b~n", [I])
       end)
    || I <- lists:seq(1,N)
  ].
```

- Running the code:

```
1> c(procs).
{ok,procs}
2> procs:hello(3).
hello world from process 1
hello world from process 2
hello world from process 3
[<0.40.0>,<0.41.0>,<0.42.0>]
```

- `when is_integer(N), N >= 0` is a guard.
  See slide 26 or Guards, Guards! in *Learn You Some Erlang*.
- `[`*Expr* `|| Var <-` *List*`]` is a **list comprehension**.
  See slide 27 or List Comprehensions in *Learn You Some Erlang*.
- `[<0.40.0>,<0.41.0>,<0.42.0>]` is the list of pids returned by `procs:hello(3)`.

Messages

- To solve tasks in parallel, the processes need to communicate.
- Message passing is fully-integrated into Erlang – it makes Erlang a simple language for getting started.
- Outline of the rest of the lecture:
  - Sending and Receiving Messages
  - Messages are asynchronous
  - Message ordering
  - Best Practices for messages

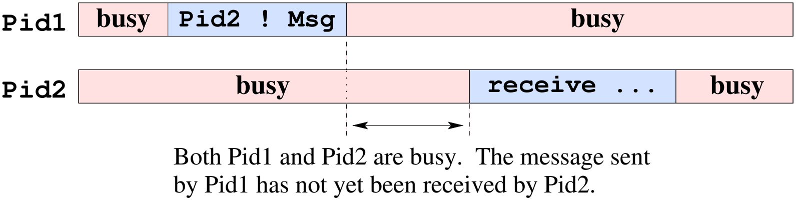
Sending and Receiving Messages

- Sending a message: `Pid ! Expr`.
  - `Expr` is evaluated, and the result is sent to process `Pid`.
  - We can send **any** Erlang term: integers, atoms, lists, tuples, ...
- Receiving a message:

```
receive
  Pattern1 -> Expr1;
  Pattern2 -> Expr2;
  ...
  PatternN -> ExprN
end
```

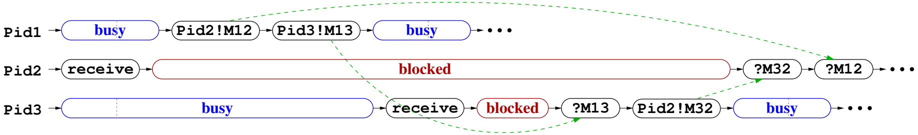
If there is a pending message for this process that matches one of the patterns,
  - The message is delivered, and the value of the `receive` expression is the value of the corresponding *Expr*.
  - Otherwise, the process blocks until such a message is received.

Message passing in Erlang is asynchronous



- Asynchronous communication lets us overlap communication with computation.
  - This can be very important for lowering the impact of high communication costs.
- But you need to be careful about synchronization.
  - If you need to guarantee that process `Pid1` does not proceed until `Pid2` receives the message.
  - Have `Pid2` send an acknowledgment back to `Pid1`, and have `Pid1` wait for the acknowledgement.
  - Conclusion: we can implement synchronous communication using asynchronous messages.

Message Ordering



- Given two processes, *Proc1* and *Proc2*, messages sent from *Proc1* to *Proc2* are received at *Proc2* in the order in which they were sent.
- Message delivery is reliable: if a process doesn't terminate, any message sent to it will eventually be delivered.
- Other than that, Erlang makes no ordering guarantees.
  - In particular, the triangle inequality is not guaranteed.
  - For example, process *Proc1* can send message *M1* to process *Proc2* and after that send message *M2* to *Proc3*.
  - Process *Proc3* can receive the message *M2*, and then send message *M3* to process *Proc2*.
  - Process *Proc2* can receive messages *M1* and *M3* in either order.

Adding two numbers using processes and messages

- The plan:
  - We'll spawn a process in the shell for adding two numbers.
  - This child process receives two numbers, computes the sum, and sends the result back to the parent.

```
add_proc(PPid)
    when is_pid(PPid) ->
  receive
    A -> receive
      B ->
        PPid ! A+B
    end
  end.

adder() ->
  MyPid = self(),
  spawn(fun() ->
    add_proc(MyPid)
  end).
```

```
3> Apid = procs:adder().
<0.44.0>
4> Apid ! 2.
2
5> Apid ! 3.
3
6> receive Sum -> Sum end.
5
```

Best Practices for Message

- Erlang has a very simple set of primitive operations for processes and communication: `spawn`, `!` (send), and `receive`. That's it!.
- Using these operations **well** requires discipline and experience. The rest of this lecture gives an overview.
  - Reactive processes and recursion: what about the call stack?
  - Tail-call elimination: an important optimization performed by the Erlang compiler. Erlang processes depend on it to avoid stack overflows.
  - Tagging messages: making sure that you receive the message you intended.
  - Time-outs: avoid hanging forever when something goes wrong.
  - Communication patterns: as *Learn You Some Erlang* said "We love messages, but we keep them secret".
- This is just an overview – you'll see more as the term goes on.

Reactive Processes and Recursion

- Often, we want processes that do more than add two numbers together. We'll use an accumulator as an example.

```
acc_proc(Tally)
    when is_integer(Tally) ->
  receive
    N when is_integer(N) ->
      acc_proc(Tally+N)/2,
    {Pid, total} ->
      Pid ! Tally,
      acc_proc(Tally)
      Tally+3;
    exit -> Tally
  end.

accumulator() ->
  spawn(fun() ->
    acc_proc(0)
  end).
```

```
7> BPid = procs:accumulator().
<0.53.0>
8> BPid ! 1.
1
9> BPid ! 2.
2
10> BPid ! 3.
3
11> BPid ! {self(), total}.
{<0.33.0>, total}
12> receive T1 -> T1 end.
6
% continued on next slide
```

- Nice, but what's up with the '/2' and '+3'?
  - It's there to illustrate a point about recursive functions.
  - See the next slide.

Reactive Processes and Recursion

- Often, we want processes that do more than add two numbers together. We'll use an accumulator as an example.

```
acc_proc(Tally)
    when is_integer(Tally) ->
  receive
    N when is_integer(N) ->
      acc_proc(Tally+N)/2,
    {Pid, total} ->
      Pid ! Tally,
      acc_proc(Tally)
      Tally+3;
    exit -> Tally
  end.

accumulator() ->
  spawn(fun() ->
    acc_proc(0)
  end).
```

```
% continued from previous slide
13> BPid ! 4.
4
14> BPid ! {self(), total}.
{<0.33.0>, total}
15> BPid ! 5.
5
16> BPid ! 6.
6
17> BPid ! {self(), total}.
{<0.33.0>, total}
18> receive T2 -> T2 end.
10
19> receive T3 -> T3 end.
21
```

- Nice, but what's up with the '`/2`'and '`+3`'?
  - It's there to illustrate a point about recursive functions.
  - See the next slide.

The many stack frames of `acc_proc`

```
acc_stack(N) ->
  AccPid = accumulator(),
  [AccPid ! I || I <- lists:seq(1, N)],
  AccPid ! {self(), total},
  receive Tally -> Tally end,
  {stack_size, Size} =
    process_info(AccPid, stack_size),
  AccPid ! exit,
  io:format(
    "N=~b, stack size = ~b, Tally=~b~n",
    [N,    Size,            Tally]).
```

N	Size
0	3
1	4
2	5
3	6
10	13
100	103
1000	1003
10000	10003
100000	100003
N	$N+3$

- Stack size grows linearly with `N`.
- Erlang is very efficient with its stack – just one Erlang “word” per call of the `acc_proc` function.
- However, if we have some kind of reactive process, we’ll eventually run out of memory for the stack.

Cleaning up acc_proc

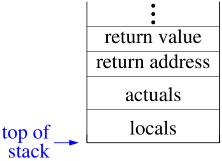
- From slide 11: "what's up with the '/2'and '+3'?"
  Let's delete that useless code.

```
acc_proc2(Tally)
    when is_integer(Tally) ->
  receive
    N when is_integer(N) ->
      acc_proc(Tally+N);
    {Pid, total} ->
      Pid ! Tally,
      acc_proc(Tally);
    exit -> ok
  end.
accumulator2() ->
  % spawns acc_proc2(0).
acc_stack2() ->
  % uses accumulate2().
```

N	Size
0	2
1	2
2	2
3	2
10	2
100	2
1000	2
10000	2
100000	2
N	2

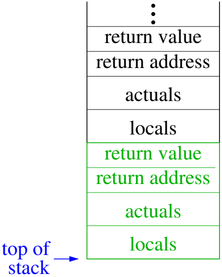
- Holy stack frames, Batman!!! What happened?

The Truth about Stack Frames



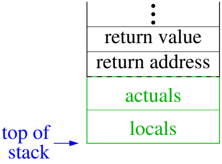
- The figure at the left shows how stack frames are often presented in first or second year CS courses.

The Truth about Stack Frames



- The figure at the left shows how stack frames are often presented in first or second year CS courses.
- When a function is called, we expect a new frame to be allocated.
  - But what happens if the caller just returns the value of the callee?
  - When the callee returns, the return value is copied, and the callee returns.

The Truth about Stack Frames



- When a function is called, we expect a new frame to be allocated.
  - But what happens if the caller just returns the value of the callee?
  - When the callee returns, the return value is copied, and the callee returns (according to introductory CS).
  - A more efficient approach is to overwrite the caller's stack frame with a new frame for the caller.
  - This is called **tail-call elimination**.
- Tail call elimination turns tail-recursive functions into while-loops.

Remarks about Tail call elimination

- Many introductory CS courses teach teach a big lie about recursion:
  - The claim is that iteration is faster than recursion.
  - With a good compiler, they can be the same.
  - You should write whichever version is **clearer**.
- Tail call elimination in various languages:
  - Erlang: mandatory – otherwise, reactive processes won't work.
  - Compilers for most functional languages (e.g. Haskell, Lisp, ML, Racket, . . .) perform tail-call elimination.
  - Java does not perform tail-call elimination – it messes with the stack based privilege management – "it seemed like a good idea at the time".
  - gcc and g++ perform tail call elimination when `-O` is given.
  - Python forbids tail-call elimination – Guido doesn't like it.

Tagging messages

- It's a very good idea to include "tags" with messages.
- This prevents your process from receiving an unintended message:
  - "Oh, I forgot that another process was going to send me that. I thought it would happen later."
  - Or, `Pid1` sends three messages to `Pid2` and you think you knew the order, but a change in the code for one process breaks the code.
- Here's an example of a "typical" tagged message:

  ```
  ToPid ! {FromPid, Tag, Data}
  ```

  Where:
  - `ToPid` – the process that will receive the message.
  - `FromPid` – the process sending the message, i.e. `self()`.
  - `Tag` – something to indicate the intended purpose of the message, often an atom.
  - `Data` – the actual content of the message.
- For example, my accumulator might be better if instead of just receiving an integer, it received

  ```
  {_FromPid, add, 2}
  ```

Time Outs – Why we need them

- Sometimes bad things happen
  - A process dies and never sends a message we expected.
  - We made a typo when tagging a message, and it doesn't match the pattern in the `receive` expression.
  - ...
- A receive can block forever if it doesn't match a message in the in-box.
- Or, we can use time-outs

```
receive
  Pattern1 -> Expr1;
  Pattern2 -> Expr2;
  ...
  PatternN -> ExprN
  after TimeOut -> % TimeOut is in milliseconds
    OopsLetsTryToRecover
end
```

Time Outs are Good

- Hanging the Erlang shell while waiting for a blocked receive can be painful.
  - We can `^C` out of the Erlang shell.
  - But I haven't found a consistent way to recover.
- We can add a time-out to the `receive` operation.
  - What should we do in the `after` clause?
  - Often, we should just print some error message and give up.
  - misc:msg_dump`(Who, PatternList)` from the CS418 Erlang library can be helpful.
    - `Who` is a string to describe what function/module/etc was attempting the receive that had the time-out.
    - `PatternList` is a list of strings – these can be cut-and-pasted from the `receive` expression. They report what patterns `Who` was looking for.
    - `msg_dump` prints the patterns and then reports all pending messages in the processes in-box.
    - This can make it easy to spot typos and other errors that led to the time-out.

Time Outs are Bad

- The value for `TimeOut` is wrong (no matter what you choose):
  - If the value is too small, then code will fail when you try to scale your application to larger problems or larger networks of machines.
  - If the value is too large, then you will spend too long waiting for time-outs.
- Conclusion:
  - Time-outs are great for debugging.
  - Time-outs can be important in production code, especially in networked applications where we are concerned about machines going down, network connectivity failing, etc.
  - If this were a course on high-reliability networked applications, we'd discuss time-outs in more detail.
  - For this course, time-outs are great for debugging, but you should be aware of their limitations.

Communications Patterns

- Communication is often the critical design consideration for parallel software.
  - We will characterize parallel algorithms by their communication patterns: trees, rings, meshes, butterflies, random, etc.
  - We will also see that the implementation of physical communication links is a key distinguishing feature of parallel architectures.
  - We will write functions that abstract communication patterns to provide a bridge between the software and implementation.
- This means you won't be writing `!` (send) or `receive` very often.
  - Unless we specifically ask you to. ☺.
  - But you'll see that this stuff is happening "under the hood" – e.g. when your code crashes and we print a backtrace.
  - You also need to make reasonable assumptions about the communication actions of our API code to get good performance.
- For more,
  - We'll be looking at trees of processes in the coming week.
  - See also *LYSE*, We love messages, but we keep them secret.

Summary

- Processes are easy to create in Erlang.
  - The `spawn` mechanism can be used to start other processors on the same CPU or on machines spread around the internet.
- Processes communicate through messages
  - Message passing is asynchronous.
  - The receiver can use patterns to select a desired message.
  - Tail-recursion is essential for implementing processes that can handle an arbitrary number of messages.
    - Your instructors lied to you if they told you that iteration is intrinsically faster than recursion.
  - Tagging and time-outs are important for writing robust code.
  - We usually abstract process creation and communication by writing APIs that support common communication patterns.
- Now, we're ready to plunge into real, parallel algorithms and software!

Preview

January 10: Reduce – The Algorithm	
Reading:	*Learn You Some Erlang*, Errors and Exceptions through A Short Visit to Common Data Structures
January 12: Reduce – The Pattern	
Reading:	Lin & Snyder, chapter 5, pp. 112–125
January 15: Scan	
Homework:	Homework 1 deadline for early-bird bonus (11:59pm) **Homework 2 goes out (due Jan. 31)** – Reduce and Scan
January 17: Reduce & Scan Examples	
Homework:	Homework 1 due **11:59pm**
January 19 & 22: Parallel Architecture	
January 24 – 31: Performance Analysis	
February 2 – 7: Sorting with Shared Memory	
February 9 – 16: TBD	
February 19 – 23: Midterm break	
February 28: Midterm	
March: Data Parallel Computing, GPUs, and CUDA	

Review Questions (1 of 2)

- How do you spawn a new process in Erlang?
- What guarantees does Erlang provide (or not) for message ordering?
- Give an example of using patterns to select messages.
- Why is it important to use a tail-recursive function for a reactive process?
  - In other words, why is it a bad idea to use a head-recursive function for a reactive process.
  - The answer isn't explicitly on the slides, but you should be able to figure it out from what we've covered.

Review Questions (2 of 2)

The `c3s_v1` and/or `c3s_v2` functions in procs.erl implement a (very inefficient) way to count the 3s in a list.

- One of `c3s_v1` or `c3s_v2` works correctly, the other does not. Compile the code and try them to determine which is which.
- Explain the differences between the two functions and how that leads to one working and the other failing.
- Implement the message flushing described in *LYSE* to show pending messages on a time-out. Use it with the `receive` operations for these count-3s functions (the receive operations are in related functions).
- How does the message-flush make the error obvious?
- Identify the recursive functions in this example.
- One of these recursive functions is not tail recursive. Which one?
- Rewrite the non tail-recursive function to be tail-recursive.

Supplementary material

- List Comprehensions
- Guards
- Tracing execution of Erlang processes

Guards

- Patterns can include guards:

  Pattern **`when`** *BoolExpr*

- This pattern matches a *Term* iff:
  - The structure of *Term* matches *Pattern*, and
  - *BoolExpr* is satisfied.
  - *BoolExpr* can consist of constants, variables, arithmetic and boolean operations, and comparisons.
  - Erlang is very restrictive about what functions you can use.
    - ★ built-in functions that have no side-effects.
    - ★ some handy ones: `element(N, Tuple)`, `is_integer(X)`, `is_list(X)`, `is_tuple(X)`, ...
- More elaborate guards can be written.
  - *BoolExpr1***,** *BoolExpr2* is roughly **andalso**.
  - *BoolExpr1***;** *BoolExpr2* is roughly **orelse**.
  - The “roughly” bit is because they handle exceptions and nesting differently. See Guards, Guards! in *LYSE* and/or Erlang Language Reference – Expressions → Guard Sequences in the Erlang documentation.
- Using guards **sensibly** can help catch errors early and make your code easier to read my making your assumptions explicit.

List Comprehensions

- The higher-order functions `map` and `filter` are used frequently in functional programs.
  - Erlang has a simple syntax for such operations.
  - It's called a **List Comprehension**.
  - `[`*Expr* `||` *Var* `<-` *List*, *Cond*, `...]`.
  - *Expr* is evaluated with *Var* set to each element of *List* that satisfies *Cond*.
- Example:

```
20>R = misc:rlist(5, 1000).
[444,724,946,502,312].
21>[X*X || X <- R, X rem 3 == 0].
[197136,97344].
```

- See also List Comprehensions in *LYSE*.

Tracing Processes

When you implement a reactive process, it can be handy to trace the execution. Here's a simple approach:

- Add an `io:format` call when entering the function and after matching each receive pattern.
- Example:

```
acc_proc(Tally) ->
  io:format("~p:  acc_proc(~b)~n", [self(), Tally]),
  receive
    N when is_integer(N) ->
      io:format("~p:  received ~b~n", [self(), N]),
      acc_proc(Tally+N);
    Msg = {Pid, total}
      io:format("~p:  received ~p~n", [self(), Msg]),
      Pid ! Tally,
      acc_proc(Tally)
  end.
```

- Try it (e.g. with the example from slide 11).
- Don't forget to delete (or comment out) such debugging output before releasing your code.

Table of Contents