CS11 – Erlang

Winter 2012–2013

Lecture 5

Registered Processes

- Normally need to know a process' PID to interact with it
- Can also register a process under a global name
  - **`register(Name, Pid)`**
    - **`Name`** must be an atom
- If **`Name`** isn't already registered, **`Pid`** is associated with **`Name`**, and **`true`** is returned
- If **`Name`** is already registered, **`register()`** reports an error
- Once a process is registered, can use **`Name`** atom directly in send operations
  - **`Name ! Expr.`**

Registered Process Example

- Echo server:

```
-module(echo).
-export([server/0]).

server() ->
    receive
        M -> io:format("~nReceived:  ~p~n", [M])
    end,
    server().
```

- Start and register the echo server:

```
Pid = spawn(echo, server, []),
register(echo_server, Pid).
```

  - Could also put **`spawn()`** call inside **`register()`** call

Registered Process Example (2)

- Once echo server is started and registered, can send messages using **`echo_server`** atom
  - **`echo_server ! {hello, world}.`**
  - Prints:
    - **`Received:  {hello, world}`**
- Note 1: This form of send can fail!
  - If atom before send-operator **`!`** is not a registered process name, then an error is reported
- Note 2: Registered procs are automatically unregistered when they terminate.

Other Registration Functions

- **`unregister(Name)`**
  - **`Name`** is an atom
  - Unregisters the process with the specified **`Name`**
  - Reports error if no process registered under **`Name`**
- **`whereis(Name)`**
  - If a process is registered under **`Name`**, returns its PID
  - Otherwise, returns the atom **`undefined`**
- **`registered()`**
  - Returns a list of the names of all registered processes

Registered Process Notes

- In distributed Erlang clusters, registered processes are local to a single node
- Be careful with registered processes in large-scale software systems!
  - To handle heavy load, should be able to parallelize every critical part of the system…
  - A registered process may represent a scalability bottleneck in your system
  - In these cases, registered process should respond to requests as quickly as possible
  - Better yet, find ways to parallelize its operation too

Linked Processes

- Two processes can be linked together
  - When one of the processes exits, the linked process receives a notification
  - Allows a process to monitor the status of another process, and handle termination signals
    - e.g. it could stop itself, or restart the dead process
- Linking is bidirectional
  - Either process can initiate the link
  - Multiple link requests are ignored
- Of course, a process can link to multiple processes

Linking Processes

- Use **`link(Pid)`** to link to another process
  - Links together processes **`self()`** and **`Pid`**
  - **`unlink(Pid)`** will remove the link
  - Note: No way to link two other processes together!
- Example: start and monitor an RSS queue

```
Queue = spawn(rss_queue, server, ["http://..."]),
link(Queue).
```

- Can also use **`spawn_link()`** to do in one step

```
Queue = spawn_link(rss_queue, server, ["http://..."]).
```

  - Same versions of **`spawn_link`** as there are **`spawn`**
    - e.g. some take funs, some take module/function/args

Process Termination

- To understand process linking, also need to understand process termination
- Processes always terminate with an *exit reason*
  - Some value indicating why the process terminated
- If a process' function returns, exit reason is the atom **`normal`**
- Errors cause an exit reason of **`{Reason,Stack}`**
  - Info about the error, as well as what code was running
  - (You have definitely seen these in the Erlang shell. ☺)
- Can use **`exit(Reason)`** BIF to terminate process
  - Can specify an appropriate reason in the call

Termination and Linking

- When two processes are linked, termination of one causes an exit signal to be sent to the other
  - Either the atom **`normal`**, or an abnormal termination signal (anything other than **`normal`**)
- Default behavior of linked processes:
  - Linked process will receive the exit signal
  - If the reason is **`normal`**, then signal is ignored
  - Otherwise, the linked process terminates with the same exit signal
    - If other processes are linked to this process, exit signal propagates to these processes as well, etc.

Trapping Exit Signals

- To handle abnormal exit signals robustly, need to trap all exit signals
  - **`process_flag(trap_exit, true)`**
  - Should be called within the process, before linking!
  - After this, process will receive messages for exit signals
    - **`{'EXIT', FromPid, Reason}`**
    - Note that **`'EXIT'`** is an atom
  - Process can handle the exit signal however it wants
    - Can restart the process that died, can exit itself, etc.

Trapping Exit Signals (2)

- Example code:

```
Pid = spawn_link(rss_queue, server, ...),
process_flag(trap_exit, true)
```

- Problems?
  - What if linked process dies before **`process_flag`** call completes?
  - Spawned process could die before you trap exit signals…
  - Since you're linked to the other process, it would kill you too
- Moral: Always trap exit signals first, before setting up links to other processes!

Sending Exit Signals to Processes

- A second form of exit function:
  - **`exit(Pid, Reason)`**
  - Sends an exit signal to the specified process, with the specified reason
  - The sending process does NOT exit!
  - Used to "fake" an exit signal, or to kill a process
- If receiving process isn't trapping exit signals
  - If reason is **`normal`** then exit signal is ignored
  - If reason is not **`normal`** then receiver will also exit with the signal/reason that was sent
- If receiver is trapping exit signals, just gets another **`{'EXIT', Pid, Reason}`** message

Untrappable Exit Signal: `kill`

- **`kill`** is an untrappable exit signal
- **`kill`** will always terminate a process
  - …regardless of whether process is trapping exit signals or not!
  - Used to handle unresponsive or runaway processes
- When process dies, **`kill`** signal does not propagate directly to linked processes!
  - Linked processes receive a **`killed`** reason, not a **`kill`** reason
  - Otherwise, too many processes could end up being killed!

One-Way Links: Monitors

- Also possible to create one-way links to other processes
  - One process **`Pid1`** monitors another process **`Pid2`**
  - Process **`Pid2`** isn't notified if **`Pid1`** terminates
- Use special method:
  - **`erlang:monitor(process, Pid)`**
  - Not automatically imported into every module
  - Must use qualified name, or explicitly import it
- Function returns a Ref (reference) value
  - Produced by **`make_ref/0`**
  - Simply a unique value generated by Erlang platform
  - (as unique as possible; does repeat after ~2^{82} calls)

Monitors (2)

- Reference value serves to identify a particular monitor link
- Like exit-signal trapping, monitoring process receives a message:
  - `{'DOWN', Ref, process, Pid, Reason}`
- A process can monitor another process multiple times
  - Each call produces a separate one-way link, with its own Ref value
  - If monitored process terminates, listening process is informed once for each monitor-call it made!

Monitors (3)

- Can unmonitor a process by calling
  - `erlang:demonitor(Ref)`

- Can specify process to monitor using its PID
- Or, if process is registered, can use its name

Throwing and Catching

- Erlang has two mechanisms for exception handling
- The simple version:
  - **`catch Expr`**
- If the expression **`Expr`** doesn't throw, **`catch`** evaluates to **`Expr`** result
- If **`Expr`** throws, catch evaluates to the value of the exception
- Throw an exception using **`throw(Expr)`** BIF
  - Argument can be any expression

Simple `throw` Example

- A simple function that throws:

```
-module(m).
-export([compute_value/1]).

compute_value(N) ->
    if
        N < 3 ->
            throw({badarg, "N must be at least 3."});
        true ->
            math:sqrt(N - 3)
    end.
```

- Function throws if passed a value less than 3

Simple `throw`/`catch` Example

- Using our new function:

```
1> m:compute_value(12).
3.00000
2> m:compute_value(2).
=ERROR REPORT====
Error in process <0.30.0> with exit value:
  {{nocatch,{badarg,"N must be at least 3."}},
  [{m,compute_value,1},{shell,exprs,6},
  {shell,eval_loop,3}]}
```

- Can catch the exception instead:

```
1> catch m:compute_value(12).
3.00000
2> catch m:compute_value(2).
{badarg,"N must be at least 3."}
```

Simple `throw`/`catch` Example (2)

- Can use **`catch`** with conditionals to figure out what happened

```
do_stuff(N, Pid) ->
    Value = catch m:compute_value(N),
    case Value of
        {badarg, _ErrMsg} ->
            ...  % handle the error
        _Else ->
            Pid ! {result,Value}
    end.
```

  - Of course, could also use **`if`** expression…

Exception Classes

- Three classes of exceptions:
  - **`throw`** – a process called **`throw()`** BIF
  - **`error`** – a runtime error occurred (e.g. **`badmatch`**)
  - **`exit`** – a process fired an exit signal
- Exception's class dictates what **`catch`** returns
  - For **`throw(Expr)`**, **`catch`** will return **`Expr`**
  - For **`error`** (runtime error), **`catch`** returns:
    - **`{'EXIT', {Reason, erlang:get_stacktrace()}}`**
  - For **`exit`** (process exit signal), **`catch`** returns:
    - **`{'EXIT', Reason}`**

`try`/`catch` Statement

- More advanced exception handling statement

```
try
    Expr1, Expr2, ...
catch
    Pattern1 -> Body1;
    Pattern2 -> Body2;
    ...
end
```

- If no exceptions, evaluates to result of the last expression in the **`try`** block
- If an exception occurs in the **`try`** expressions:
  - First matching **`catch`** pattern is evaluated; **`try`**-block evaluates to result of corresponding **`catch`**-body
  - If no **`catch`** pattern matches then exception from **`try`** clause propagates out of entire **`try`**/**`catch`** statement

`try`/`catch` Statement (2)

- Additional exception-matching capabilities:

```
try
    Expr1, Expr2, ...
catch
    [Class1:]Pattern1 [when Seq1] -> Body1;
    [Class2:]Pattern2 [when Seq2] -> Body2;
    ...
end
```

- Can optionally specify exception class in **`catch`** patterns
  - Either **`throw`**, **`error`**, or **`exit`**
- Can also specify guard sequences, as usual

`try`/`catch` Statement (3)

- Additional **`try`**-expression matching features:

```
try Expr of
    Pattern1 [when Seq1] -> Body1;
    Pattern2 [when Seq2] -> Body2;
    ...
catch
    [Class1:]Pattern1 [when Seq1] -> Body1;
    [Class2:]Pattern2 [when Seq2] -> Body2;
    ...
end
```

- Identical to **`case`** statement, but also includes exception handling

`try`/`catch` Statement (4)

- Additional **`try`**-expression matching features:

```
try Expr of
    Pattern1 [when Seq1] -> Body1;
    Pattern2 [when Seq2] -> Body2;
    ...
catch
    [Class1:]Pattern1 [when Seq1] -> Body1;
    [Class2:]Pattern2 [when Seq2] -> Body2;
    ...
end
```

- If no **`try`** clause matches, **`try_clause`** error is thrown
  - **Note**: Can't catch this error with a **`catch`** clause!
- **`try`** only catches exceptions thrown by the **`try`**-exprs
  - If **`catch`** clause throws, propagates out of entire statement

`try`/`catch` Statement (5)

- Can specify final processing after normal or abnormal completion:

```
try ...
catch ...
after
    Body
end
```

  - Very useful for cleaning up resources, regardless of whether exception was thrown
- **`of`**, **`catch`**, and **`after`** clauses are all optional
  - Must have either a **`catch`** or an **`after`** clause

This Week's Assignment

- New features for your RSS aggregator!
  - Retrieve RSS feeds from actual web servers
  - Aggregate multiple feeds into a single queue
- RSS queues have two modes of operation:
  - Get feed items from a URL
  - Get feed items from other RSS queues
- First mode requires talking to the Interwebs
- Can use Erlang `http` module to retrieve URLs
  - May fail with various errors, so we need error reporting and handling!

The `http` Module

- A simple function: **`http:request(Url)`**
- On success, **`{ok, Result}`** is returned
  - Result is a composite value containing HTTP status code, response headers, and response body
  - See docs for details of result!
- Some failures cause **`{error, Reason}`** to be thrown
- Not all **`ok`** results are acceptable!
  - e.g. may get an **`ok`** result, with a 404 status code!
  - In these cases, need to report the issue using an exception or an exit signal

Error Logging

- Will have a number of processes running…
  - Need to know what they are actually doing!
- Incorporate logging into your code this week
- Erlang provides an **`error_logger`** module
  - Provides an error-logging server process
    - Registered under name **`error_logger`**
  - Also helper functions to report info messages, warnings, and errors
    - Very similar to **`io:format/2`** capabilities
    - These funcs send messages to **`error_logger`** process
- I will give you some helpful logging macros

Next Time!

- Quite possibly the coolest feature of Erlang
- The ability to create *generic servers*
  - Abstract out all of the server-specific details
    - e.g. what messages to handle, how to handle them
  - Provide a generic server behavior, and then plug in specific implementation details
- Gets to the core ideas behind the OTP
  - Facilitates very powerful and extensible systems
  - Very cool stuff!!!