CS11 – Erlang

Winter 2008–2009

Lecture 6

Today's Project: Calculator Server

- Create a simple calculator server, and register the process under a specific name
- Send messages to the calculator to perform simple arithmetic operations
  - **`{clear}`** resets calculator to 0
  - **`{set, Value}`** sets calculator to specified value
  - **`{get}`** retrieves current value
  - **`{add, Value}`** adds **`Value`** to current value, and then returns the result
  - etc.
- Calculator needs to send back results too…
  - Include sender's PID in messages to server

Calculator Implementation

```
-module(calc).
-export([start/0, start/1, send/2]).

% Server has one piece of state - the current result.
server(Value) when is_number(Value) ->
    receive
        {From, {clear}} ->
            From ! 0,
            server(0);
        {From, {set, Num}} when is_number(Num) ->
            From ! Num,
            server(Num);
        {From, {add, Num}} when is_number(Num) ->
            NewValue = Value + Num,
            From ! NewValue,
            server(NewValue);
        ...
```

Calculator Implementation (2)

```
...

start() ->
    spawn(fun () -> server(0) end).

start(Name) ->
    register(Name, start()).

% Helper to wrap the request/response sequence.
send(Calc, Msg) when (is_pid(Calc) orelse is_atom(Calc)),
                     is_tuple(Msg) ->
    Calc ! {self(), Msg},
    receive
        Result -> Result
    after 1000 ->
        {error, "Calculator not responding."}
    end.
```

Using Our Calculator

- Compile and run our calculator:

```
1> c(calc).
{ok,calc}
2> calc:start(calculon).
true
3> calc:send(calculon, {set, 36}).
36
4> calc:send(calculon, {add, 23}).
59
5> calc:send(calculon, {clear}).
0
```

Generic Server Components

- A lot of our calculator server is generic
  - Doesn't really depend on the specific server we are implementing
- Some generic components:
  - Starting and registering the server
  - Helper for exchanging messages with the server
- Even the server loop can be made generic
  - Actual state being managed is server-specific
  - What incoming messages are handled, what responses are returned
  - By imposing some standards, can make this generic

Generic Server Loop

- A more generic server loop:

```
loop(State) ->
    receive
        {From, Request} ->
            {Response, NewState} =
                handle(Request, State),
            From ! Response,
            loop(NewState)
    end.
```

- Now all server-specific code is contained within this **`handle()`** function
  - Just have to find a way to pass this to the generic server component…

Calling Erlang Functions

- So far, have called functions in different modules like this:

```
io:format("Hello world!~n").
```

  - First part is module name: `io`
  - Second part is function name: `format`
- Can actually use any expression that evaluates to valid module and function names
  - Remember: module and function names are atoms

```
Mod = io,
Func = format,
Mod:Func("Hello world!~n").
```

Calling Erlang Functions (2)

- In fact, can also specify **`mod:func`** as a tuple:

```
Func = {io, format},
Func("Hello world!~n").
```

- Can use this calling mechanism to create generic server components
  - Generic server expects a specific set of callback functions to be provided
    - e.g. **`handle(Request, State)`**
  - At startup, generic server takes a “callback module” argument
    - Callback module provides server-specific capabilities
    - Generic server provides commonly used server features

Refactoring our Calculator

- First, factor out generic server components:

```
-module(server).
-export([start/2, rpc/2]).

% Generic server only works with registered processes
%   Name  = registered name for the process
%   Mod   = the callback module
%   State = the current server state
loop(Name, Mod, State) ->
    receive
        {From, Request} ->
            {Response, NewState} =
                Mod:handle(Request, State),
            From ! {Name, Response},
            loop(Name, Mod, NewState)
    end.
```

Generic Server (2)

- Now, implement **`server:start/2`**
- Problem:
  - Need to generate “initial state” to pass to generic server-loop function
- Callback module provides an **`init()`** function that returns the initial server state

```
% Generic "start-server" function - construct initial
% state, then start server and register the server.
start(Name, Mod) ->
    State0 = Mod:init(),
    Pid = spawn(fun() -> loop(Name, Mod, State0) end),
    register(Name, Pid).
```

Generic Server (3)

- Finally, helper function to make synchronous calls to the server

```
% Make a remote call to the server, then
% return the response.
rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, Response} -> Response
    after 1000 ->
        {error, timeout}
    end.
```

- **rpc** = "remote procedure call"
  - Common name for calling between processes, regardless of whether on same machine or not

Calculator Callback Module

- Now, factor out calculator-specific code

```
-module(calc2).
-export([init/0, handle/2]).

% Callback to return the initial state.
init() -> 0.

% Handler for various messages:
%   {Request, State} -> {Response, NewState}
handle({clear}, _OldVal) ->
    {0, 0};
handle({set, Num}, _OldVal) with is_number(Num) ->
    {Num, Num};
handle({add, Num}, OldVal) with is_number(Num) ->
    NewVal = OldVal + Num,
    {NewVal, NewVal};
...
```

New Calculator!

- Try out our new calculator:

```
1> server:start(calculon2, calc2).
true
2> server:rpc(calculon2, {set, 15}).
15
3> server:rpc(calculon2, {add, 24}).
39
4> server:rpc(calculon2, {clear}).
0
```

- Calculator code is *entirely sequential*
  - No message-passing or process-management code
  - Generic server framework takes care of these details
  - Can write servers without knowing anything about these details!

Using our Calculator

- Go ahead and use the new calculator:

```
1> server:start(calculon2, calc2).
true
2> server:rpc(calculon2, {set, 15}).
15
3> server:rpc(calculon2, {add, "24"}).
=ERROR REPORT==== ...
Error in process ...
{error,timeout}
4>
```

- We sent some bad data and the server died ☹

Add Error Handling to Server Loop

```
loop(Name, Mod, State) ->
    receive
        {From, Request} ->
            try Mod:handle(Request, State) of
                {Response, NewState} ->
                    From ! {Name, ok, Response},
                    loop(Name, Mod, NewState)
            catch
                _:Reason ->
                    error_logger:error_msg(
                        "Error in ~p server \"~p\":  ~p",
                        [Mod, Name, Reason]),
                    From ! {Name, error, Reason},
                    % Loop with previous state.
                    loop(Name, Mod, State)
            end
    end.
```

Updated Server Framework

- Now our server can handle errors
  - If **`handle/2`** callback function crashes, it doesn't kill the server
  - Just report error to client, then go on with previous server state
    - Provides atomic transaction semantics
  - (Also need to update **`server:rpc/2`** func to look for error response, possibly throw the error, etc.)
- We upgraded the server's capabilities without any changes to the callback module!
  - All specific servers benefit from upgrades to generic framework

Erlang/OTP Behaviors

- In large, multi-server systems, most servers contain generic code and specific code
- Factor server code into:
  - A behavior module (the generic part)
  - A callback module (the specific part)
- The OTP provides several generic behaviors:
  - **`gen_server`** for client/server interactions
  - **`gen_fsm`** for finite state machines
  - **`gen_event`** for event handling
  - **`supervisor`** for managing other servers
- Each behavior has its own capabilities, and its own callback interface
  - Just implement the callbacks, and you're done!

The `gen_server` Behavior

- The **`gen_server`** behavior is for implementing a server for client/server interactions
- Callback module provides these functions:

`Module:init/1`
  - Returns initial state for server

`Module:handle_call/3`
  - Handles a synchronous request/response

`Module:handle_cast/2`
  - Handles an asynchronous message

`Module:handle_info/2`
  - Handles process-exit signals from linked processes

`Module:terminate/2`
  - Handles server shutdown

`Module:code_change/3`
  - Handles code-upgrades on a running server

Behavior Callback Functions

- Does Erlang actually help verify that the callback module provides the necessary functions?!
- Yes, with the **`-behaviour()`** module attribute!
  - (Note the non-US spelling…)

```
-module(calc3).
-behaviour(gen_server).
-export([init/1, handle_call/3, handle_cast/2,
         terminate/2, code_change/3]).

% our server implementation
...
```

- Compiler will report warnings if callback module is missing any expected functions

Starting a Generic Server

- **`gen_server`** provides several start functions
  - **`start(Module, Args, Options)`**
    - Starts the **`gen_server`** process
  - **`start_link(Module, Args, Options)`**
    - Starts and then links to the **`gen_server`** process
- **`Module`** specifies the callback module
- **`Args`** is an arbitrary single term, passed to **`Module:init/1`** callback
- **`Options`** are **`gen_server`** options
  - See **`gen_server`** module docs for details

Starting a Generic Server (2)

- **`Module:init(Args)`** can return several values
  - **`{ok, State}`**
    - Initialization succeeded; **`State`** is initial server state
  - **`{ok, State, Timeout}`**
    - Same as above
    - **`Timeout`** value (integer, ms) tells **`gen_server`** to notify callback module if no messages arrive before timeout
      - Calls **`Module:handle_info/2`** callback function
  - **`{stop, Reason}`**
    - Aborts server initialization, with specified reason
  - **`ignore`**
    - Aborts server initialization, with no reason given

Starting a Generic Server (3)

- **`gen_server`** start functions return:
  - **`{ok, Pid}`** if server was started successfully
  - **`{error, Reason}`** if **`Module:init/1`** returns **`{stop, Reason}`**
  - **`ignore`** if **`Module:init/1`** returns **`ignore`**
- Start functions don't return until **`Module:init/1`** actually completes
  - Ensures no chance of race-conditions at startup
- Note:
  - Also versions of **`gen_server`** start functions that register server under a name (local or global)

Handling Incoming Calls

- **`gen_server:call(ServerRef, Request)`**
  - Implements synchronous request/response interaction with server
  - **`gen_server`** forwards request to callback module
- **`Module:handle_call(Request, From, State)`**
  - **`Request`** is same as passed to **`call/2`**
  - **`From`** is a **`{Pid, Tag}`** tuple
    - **`Tag`** is a unique Ref value used to identify this request
  - **`State`** is current state of server
- As before, **`handle_call`** can process the incoming request
  - Then, response from **`handle_call`** tells **`gen_server`** what to do next

Handling Incoming Calls (2)

- **`handle_call`** result tells **`gen_server`** what to do:
  - **`{reply, Reply, NewState}`**
  - **`{reply, Reply, NewState, Timeout}`**
    - **`Reply`** value is sent back to caller
    - **`NewState`** is used for next iteration of call
    - **`Timeout`** (integer, ms) tells server to notify callback if no messages received before timeout
  - **`{noreply, NewState}`**
  - **`{noreply, NewState, Timeout}`**
    - Used when server can't return response at end of function
    - Instead, server uses **`gen_server:reply/2`** to respond
  - **`{stop, Reason, NewState}`**
  - **`{stop, Reason, Reply, NewState}`**
    - Tells server to shut down. **`Module:terminate/1`** is called.
    - **`Reply`** is returned to caller, if specified

Asynchronous Calls

- **`gen_server:cast(ServerRef, Request)`**
  - Sends an asynchronous message to the server
  - Immediately returns **`ok`** without waiting for response
  - **`gen_server`** forwards message to callback module
- **`Module:handle_cast(Request, State)`**
  - **`Request`** is same as passed to **`cast/2`**
  - **`State`** is current state of server
- Return value tells **`gen_server`** what to do:
  - **`{noreply, NewState}`** (update state)
  - **`{noreply, NewState, Timeout}`** (specify timeout)
  - **`{stop, Reason, NewState}`** (stop the server)
  - Note: **`handle_cast`** doesn't return **`{reply, ...}`**

Termination, Code-Changes

- When server terminates, **`gen_server`** calls the **`Module:terminate(Reason,State)`** callback
  - Server can save its state to persistent storage, etc.
  - Return value of **`terminate`** is ignored
- At this point, you probably won't do anything clever for this function
- Same for **`Module:code_change/3`** function
  - When a server's code is upgraded, **`gen_server`** calls **`Module:code_change(OldVsn,State,Extra)`**
  - Allows newly loaded server code to migrate its old state to the current version

Calculator, `gen_server` Style

- Provide a **`calc3`** callback module

```
-module(calc3).
-behaviour(gen_server).
-export([init/1, handle_call/3, handle_cast/2,
         handle_info/2, terminate/2, code_change/3]).
```

- Implement **`init/1`** and **`handle_call/3`**
- Use stubs for functions we don't care about
  - Examples:

```
handle_cast(_Msg, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
```

gen_server Calculator (2)

- Functions we provide:

```
% Server state is a single number.
init([]) -> 0.

% Handle incoming requests.
handle_call({clear}, _From, _Value) ->
    {reply, 0, 0};
handle_call({set, Num}, _From, _Value)
        when is_number(Num) ->
    {reply, Num, Num};
handle_call({add, Num}, _From, Value)
        when is_number(Num) ->
    NewValue = Value + Num,
    {reply, NewValue, NewValue};
    ...
```

Using the gen_server Calculator

- Once compiled, can use **gen_server** to interact with calculator

```
1> gen_server:start({local,calculon},calc3,[],[]).
{ok,<0.32.0>}
2> gen_server:call(calculon, {set, 19}).
19
3> gen_server:call(calculon, {add, 5}).
24
4> gen_server:call(calculon, {clear}).
0
```

- Note: using the "registered server" version of **gen_server:start** to name server "calculon"

Other `gen_server` Notes

- Gets really annoying to use the **`gen_server`** functions to interact to your server!

```
gen_server:start
gen_server:start_link
gen_server:call
gen_server:cast
```

- Normally, callback modules provide helper functions to handle common tasks

```
start(Name) ->
    gen_server:start({local,Name},?MODULE,[],[]).
add(Name, Num) when is_number(Num) ->
    gen_server:call(Name, {add,Num}).
```

  - Also helps encapsulate server's RPC message-protocol

Other `gen_server` Notes (2)

- With helper functions, interaction becomes very easy:

```
1> calc3:start(calculon).
{ok,<0.32.0>}
2> calc3:set(calculon, 15).
15
3> calc3:add(calculon, 22).
37
4> calc3:clear(calculon).
0
```

- Callback modules may provide many helper functions like these…

Other `gen_server` Notes (3)

- If a server should receive process-exit signals, do this in **`Module:init/1`** callback
  - Doesn't happen automatically!
  - Notifications will arrive via **`handle_info`** callback

References

- Programming Erlang by Joe Armstrong
  - Chapter 16: OTP Introduction
  - A *fantastic* overview of these ideas
- OTP Design Principles on Erlang website
  - http://www.erlang.org/doc/design_principles/users_guide.html
  - First two chapters cover behaviors, and **`gen_server`**
  - Want to define your own behaviors? See chapter 6!
- Erlang API docs for **`gen_server`** module
  - http://www.erlang.org/doc/man/gen_server.html
  - Very detailed descriptions of callback function arguments and return values