5

Process Design Patterns

Processes in Erlang systems can act as gateways to databases, handle protocol stacks, or manage the logging of trace messages. Although these processes may handle different requests, there will be similarities in how these requests are handled. We call these similarities *design patterns*. In this chapter, we are going to cover the most common patterns you will come across when working with Erlang processes.

The *client/server* model is commonly used for processes responsible for a *resource* such as a list of rooms, and *services* which can be applied on these resources such as booking a room or viewing its availability. Requests to this server will allow clients (usually implemented as Erlang processes) to access these resources and services.

Another very common pattern deals with *finite state machines*, also referred to as FSMs. Imagine a process handling events in an instant messaging (IM) session. This process, or finite state machine as we should call it, will be in one of three states. It could be in an `offline` state, where the session with the remote IM server is being established. It could be in an `online` state, enabling the user to send and receive messages and status updates. And finally, if the user wants to remain online but not receive any messages or status updates, it could be in a `busy` state. State changes are triggered through process messages we call *events*. An IM server informing the FSM that the user is logged on successfully would cause a *state transition* from the `offline` state to the `online` state. Events received by the FSM do not necessarily have to trigger state transitions. Receiving an instant message or a status update would keep the FSM in an `online` state while a logout event would cause it to go from an `online` or `busy` state to the `offline` state.

The last pattern we will cover is the *event handler*. Event handler processes will receive messages of a specific type. These could be trace messages generated in your program or stock quotes coming from an external feed. Upon receiving these events, you might want to perform a set of actions such as triggering an SMS (Short Message Service message) or sending an email if certain conditions are met, or simply logging the time, the quote, and the price in a file.

Many Erlang processes will fall into one of these three categories. In this chapter, we will look at examples of process design patterns, explaining how they can be used to program

client/servers, finite state machines, and event handlers. An experienced Erlang programmer will recognize these patterns in the design phase of the project and use libraries and templates which are part of the OTP framework. For the time being, we will use Erlang without the OTP framework. We will introduce OTP behaviors in Chapter 12.

Client/Server Models

Erlang processes can be used to implement client/server solutions, where both clients and servers are represented as Erlang processes. A server could be a FIFO queue to a printer, a window manager, or a file server. The resources it handles could be a database, a calendar, or a finite list of items such as rooms, books, or radio frequencies. Clients use these resources by sending the servers requests to print a file, update a window, book a room, or use a frequency. The server receives the request, handles it, and responds with an acknowledgment and a return value if the request was successful, or with an error if the request did not succeed (see Figure 5-1).

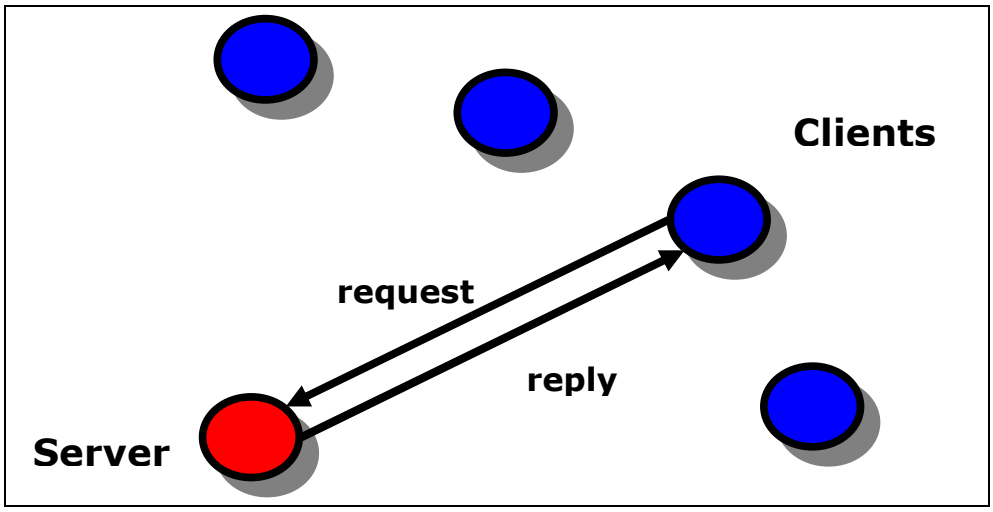

Figure 5-1. The client/server model

When implementing client/server behavior, clients and servers are represented as Erlang processes. Interaction between them takes place through the sending and receiving of messages. Message passing is often hidden in functional interfaces, so instead of calling

```
printerserver ! {print, File}
```

a client would call

```
printerserver:print(File)
```

This is a form of *information hiding*, where we do not make the client aware that the server is a process, that it could be registered, and that it might reside on a remote computer. Nor do we expose the message protocol being used between the client and the server, keeping the interface between them safe and simple. All the client needs to do is call a function and expect a return value.

Hiding this information behind a functional interface has to be done with care. The message response times will differ if the process is busy or running on a remote machine. Although this should in most cases not cause any issues, the client needs to be aware of it

and be able to cope with a delay in response time. You also need to factor in that things can go wrong behind this function call. There might be a network glitch, the server process might crash, or there might be so many requests that the server response times become unacceptable.

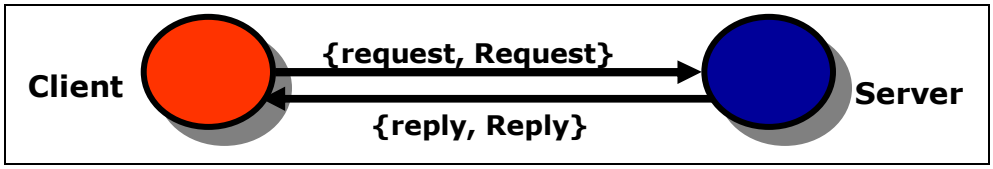

Figure 5-2. Synchronous client/server requests

If a client using the service or resource handled by the server expects a reply to the request, the call to the server has to be *synchronous*, as in figure 5-2. If the client does not need a reply, the call to the server can be *asynchronous*. When you encapsulate synchronous and asynchronous calls in a function call, asynchronous calls commonly return the atom `ok`, indicating that the request was sent to the server. Synchronous calls will return the value expected by the client. These return values usually follow the format `ok`, `{ok, Result}`, or `{error, Reason}`.

A Client/Server Example

Enough with the theory! So that you understand what we are talking about, let's walk through a client/server example and test it in the shell. This server is responsible for managing radio frequencies on behalf of its clients, the mobile phones connected to the network. The phone requests a frequency whenever a call needs to be connected, and releases it once the call has terminated (see Figure 5-3).

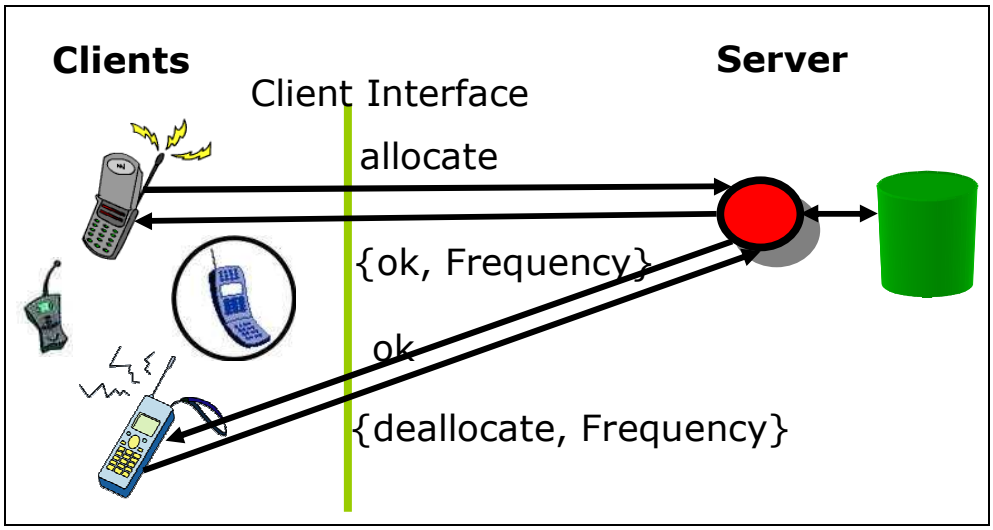

Figure 5-3. A frequency server

When a cell phone has to set up a connection to another subscriber, it calls the `frequency:allocate()` client function. This call has the effect of generating a synchronous message which is sent to the server. The server handles it and responds with either a message containing an available frequency or an error if all frequencies are being

used. The result of the `allocate/0` call will therefore be either `{ok, Frequency}` or `{error, no_frequencies}`.

Through a functional interface, we hide the message-passing mechanism, the format of these messages, and the fact that the frequency server is implemented as a registered Erlang process. If we were to move the server to a remote host, we could do so without having to change the client interface.

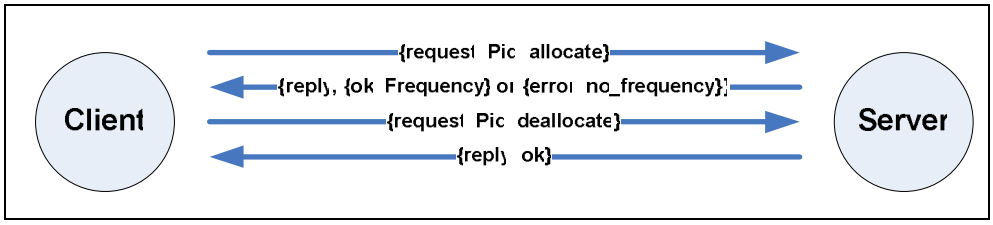

Figure 5-4. Frequency server message sequence diagram

When the client has completed its phone call and releases the connection, it needs to deallocate the frequency so that other clients can reuse it. It does so by calling the client function `frequency:deallocate(Frequency)`. The call results in a message being sent to the server. The server can then make the frequency available to other clients and responds with the atom `ok`. The atom is sent back to the client and becomes the return value of the `deallocate/1` call. The message sequence diagram of this example is shown in figure 5-4.

The code for the server is in the `frequency` module. Here is the first part:

```
-module(frequency).
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/0]).

%% These are the start functions used to create and
%% initialize the server.

start() ->
  register(frequency, spawn(frequency, init, [])).

init() ->
  Frequencies = {get_frequencies(), []},
  loop(Frequencies).

% Hard Coded
get_frequencies() -> [10,11,12,13,14,15].
```

The `start` function spawns a new process which starts executing the `init` function in the `frequency` module. The `spawn` returns a pid which is passed as the second argument to the `register` BIF. The first argument is the atom `frequency`, which is the alias with which the process is registered. This follows the convention of registering a process with the same name as the module in which it is defined.

> Remember that when spawning a process, you have to export the `init/0` function as it is used by the `spawn/3` BIF. We have put this function in a separate `export` clause to distinguish it from the client functions, which are supposed to be called from other modules. Calling

> `frequency:init()` from anywhere in your code would be considered very bad practice, and should not be done.

The newly spawned process starts executing in the `init` function. It creates a tuple consisting of the available frequencies, retrieved through the `get_frequencies/0` call, and a list of the allocated frequencies, initially given by the empty list as the server has just been started. The tuple, which forms what we call the *state* or *loop data*, is bound to the `Frequencies` variable and passed as an argument to the receive-evaluate function, which in this example we've called `loop/1`.

In the `init/0` function, we use the variable `Frequencies` for readability reasons, but nothing is stopping us from creating the tuple directly in the `loop/1` call, as in the call `loop({get_frequencies(), []})`.

Here is how the *client functions* are implemented:

```
%%  The client Functions

stop()           -> call(stop).
allocate()       -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).

%% We hide all message passing and the message
%% protocol in a functional interface.

call(Message) ->
  frequency ! {request, self(), Message},
  receive
    {reply, Reply} -> Reply
  end.
```

Client and supervisor[1] processes can interact with the frequency server using what we refer to as *client functions*. These exported functions include `start`, `stop`, `allocate`, and `deallocate`. They call the `call/1` function, passing the message to be sent to the server as an argument. This function will encapsulate the message protocol between the server and its clients, sending a message of the format `{request, Pid, Message}`. The atom `request` is a tag in the tuple, `Pid` is the process identifier of the calling process (returned by calling the `self()` BIF in the calling process), and `Message` is the argument originally passed to the `call/1` function.

When the message has been sent to the process, the client is suspended in the `receive` clause waiting for a response of the format `{reply, Reply}`, where the atom `reply` is a tag and the variable `Reply` is the actual response. The server response is pattern-matched and the contents of the variable `Reply` become the return value of the client functions.

Pay special attention to how message passing and the message protocol have been abstracted to a format independent of the action relating to the message itself; this is what we referred to earlier as *information hiding*, allowing the details of the protocol and the message structure to be modified without affecting any of the client code.

[1] We will cover supervisors in the next chapter.

Now that we have covered the code to start and interact with the frequency server, let's take a look at its receive-evaluate loop:

```
%% The Main Loop

loop(Frequencies) ->
  receive
   {request, Pid, allocate} ->
     {NewFrequencies, Reply} = allocate(Frequencies, Pid),
      reply(Pid, Reply),
      loop(NewFrequencies);
    {request, Pid , {deallocate, Freq}} ->
      NewFrequencies = deallocate(Frequencies, Freq),
      reply(Pid, ok),
      loop(NewFrequencies);
    {request, Pid, stop} ->
      reply(Pid, ok)
  end.

reply(Pid, Reply) ->
  Pid ! {reply, Reply}.
```

The `receive` clause will accept three kinds of requests originating from the client functions, namely `allocate`, `deallocate`, and `stop`. These requests follow the same format defined in the `call/1` function, that is, `{request, Pid, Message}`. The `Message` is pattern-matched in the expression and used to determine which clause is executed. This, in turn, determines the internal functions which are called. These internal functions will return the new loop data, which in our example consists of the new lists of available and allocated frequencies, and where needed, a reply to send back to the client. The client pid, sent as part of the request, is used to identify the calling process and is used in the `reply/2` call.

Assume a client wants to *initiate* a call. To do so, it would request a frequency by calling the `frequency:allocate()` function. This function sends a message of the format `{request, Pid, allocate}` to the frequency server, pattern matching in the first clause of the `receive` statement. This message will result in the server function `allocate(Frequencies, Pid)` being called, where `Frequencies` is the loop data containing a tuple of allocated and available frequencies. The `allocate` function (defined shortly) will check whether there are any available frequencies:

- If so, it will return the updated loop data, where the newly allocated frequency has been moved from the available list and stored together with the pid in the list of allocated ones. The reply sent to the client is of the format `{ok, Frequency}`.
- If no frequencies are available, the loop data is returned unchanged and the `{error, no_frequency}` message is returned as a reply.

The `Reply` is sent to the `reply(Pid, Message)` call, which formats it to the internal client/server message format and sends it back to the client. The function then calls `loop/1` recursively, passing the new loop data as an argument.

Deallocation works in a similar way. The client function results in the message `{request, Pid, deallocate}` being sent and matched in the second clause of the `receive` statement. This makes a call to `deallocate(Frequencies, Frequency)` and the `deallocate` function moves the `Frequency` from the

allocated list to the deallocated one, returning the updated loop data. The atom `ok` is sent back to the client, and the `loop/1` function is called recursively with the updated loop data.

If the `stop` request is received, `ok` is returned to the calling process and the server terminates, as there is no more code to execute. In the previous two clauses, `loop/1` was called in the final expression of the `case` clause, but not in this case.

We complete this system by implementing the allocation and deallocation functions:

```
%% The Internal Help Functions used to allocate and
%% deallocate frequencies.

allocate({[], Allocated}, _Pid) ->
    {{[], Allocated}, {error, no_frequency}};
allocate({[Freq|Free], Allocated}, Pid) ->
    {{Free, [{Freq, Pid}|Allocated]}, {ok, Freq}}.

deallocate({Free, Allocated}, Freq) ->
    NewAllocated=lists:keydelete(Freq, 1, Allocated),
    {[Freq|Free],  NewAllocated}.
```

The `allocate/2` and `deallocate/2` functions are local to the `frequency` module, and are what we refer to as *internal help* functions:

- If there are no available frequencies, `allocate/2` will pattern-match in the first clause, as the first element of the tuple containing the list of available frequencies is empty. This clause returns the `{error, no_frequency}` tuple alongside the unchanged loop data.
- If there is at least one available frequency, the second clause will match successfully. The frequency is removed from the list of available ones, paired up with the client pid, and moved to the list of allocated frequencies.

The updated frequency data is returned by the `allocate` function. Finally, `deallocate` will remove the newly freed frequency from the list of allocated ones using the `lists:keydelete/3` library function and concatenate it to the list of available frequencies.

This frequency allocator example has used all of the key sequential and concurrent programming concepts we have covered so far. They include pattern matching, recursion, library functions, process spawning, and message passing. Spend some time making sure you understand them. You should test the example using the debugger and the process manager, following the message passing protocols between the client and server and the sequential aspects of the loop function. You can see an example of the frequency allocator in action now:

```
1> c(frequency).
{ok,frequency}
2> frequency:start().
true
3> frequency:allocate().
{ok,10}
4> frequency:allocate().
{ok,11}
5> frequency:allocate().
{ok,12}
```

```
6> frequency:allocate().
{ok,13}
7> frequency:allocate().
{ok,14}
8> frequency:allocate().
{ok,15}
9> frequency:allocate().
{error,no_frequency}
10> frequency:deallocate(11).
ok
11> frequency:allocate().
{ok,11}
12> frequency:stop().
ok
```

A Process Pattern Example

Now let's look at similarities between the client/server example we just described and the process skeleton we introduced in Chapter 4. Picture an application, either a web browser or a word processor, which handles many simultaneously open windows centrally controlled by a window manager. As we aim to have a process for each truly concurrent activity, spawning a process for every window is the way to go. These processes would probably not be registered, as many windows of the same type could be running concurrently.

After being spawned, each process would call the `initialize` function which draws and displays the window and its contents. The return value of the `initialize` function contains references to the widgets displayed in the window. These references are stored in the `state` variable and are used whenever the window needs updating. The `state` variable is passed as an argument to a tail-recursive function that implements the receive-evaluate loop.

In this loop function, the process waits for events originating in or relating to the window it is managing. It could be a user typing in a form or choosing a menu entry, or an external process pushing data which needs to be displayed. Every event relating to this window is translated to an Erlang message and sent to the process. The process, upon receiving the message, calls the `handle` function, passing the message and state as arguments. If the event were the result of a few keystrokes typed in a form, the `handle` function might want to display them. If the user picked an entry in one of the menus, the `handle` function would take appropriate actions in executing that menu choice. Or if the event was caused by an external process pushing data, possibly an image from a webcam or an alert message, the appropriate widget would be updated. The receipt of these events in Erlang would be seen as a generic pattern in all processes. What would be considered specific and change from process to process is how these events are handled.

Finally, what if the process receives a `stop` message? This message might have originated from a user picking the Exit menu entry or clicking the Destroy button, or from the window manager broadcasting a notification that the application is being shut down. Regardless of the reason, a `stop` message is sent to the process. Upon receiving it, the process calls a `terminate` function which destroys all of the widgets, ensuring that they are no longer displayed. After the window has been shut down, the process terminates because there is no more code to execute.

Look at the following process skeleton. Could you not fit all of the specific code into the `initialize/1`, `handle_msg/2`, and `terminate/1` functions?

```
-module(server).
-export([start/2, stop/1, call/2]).
-export([init/1]).

start(Name, Data) ->
  Pid = spawn(generic_handler, init,[Data])
  register(Name, Pid), ok.

stop(Name) ->
  Name ! {stop, self()},
  receive {reply, Reply} -> Reply end.

call(Name, Msg) ->
  Name ! {request, self(), Msg},
  receive {reply, Reply} -> Reply end.

reply(To, Msg) ->
  To ! {reply, Msg}.

init(Data) ->
  loop(initialize(Data)).

loop(State) ->
  receive
    {request, From, Msg} ->
      {Reply,NewState} = handle_msg(Msg, State),
      reply(From, Reply),
      loop(NewState);
    {stop, From}  ->
      reply(From, terminate(State))
  end.

initialize(…)   -> …
handle_msg(…,…) -> …
terminate(…)    -> …
```

Using the generic code in the preceding skeleton, let's go through the GUI example one last time:

- The `initialize` function draws the window and displays it, returning a reference to the widget which gets bound to the `state` variable.
- Every time an event arrives in the form of an Erlang message, the event is taken care of in the `handle_msg` function. The call takes the message and the state as arguments and returns an updated `State` variable. This variable is passed to the recursive `loop` call, ensuring that the process is kept alive. Any reply is also sent back to the process where the request originated.
- If the `stop` message is received, `terminate` is called, destroying the window and all the widgets associated with it. The `loop` function is not called, allowing the process to terminate normally.

Finite State Machines

Erlang processes can be used to implement finite state machines. A finite state machine, or FSM for short, is a model which consists of a finite number of states and events. You can think of an FSM as a model of the world which will contain abstractions from the details of the real system. At any one time, the FSM is in a specific state. Depending on the incoming event and the current state of the FSM, a set of actions and a transition to a new state will occur (see Figure 5-5).

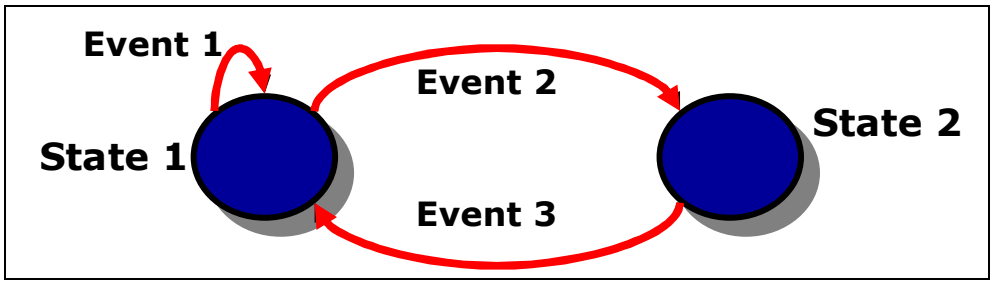

Figure 5-5. A finite state machine

In Erlang, each state is represented as a tail-recursive function and each event is represented as an incoming message. When a message is received and matched in a `receive` clause, a set of actions are executed. These actions are followed by a state transition achieved by calling the function corresponding to the new state.

An FSM Example

As an example, think of modeling a fixed-line phone as a finite state machine (see Figure 5-6). The phone can be in the `idle` state, when it is plugged in and waiting either for an incoming phone call or for a user to take it off the hook. If you receive an incoming call from your aunt,[2] the phone will start ringing. Once it has started ringing, the state will change from `idle` to `ringing` and will wait for one of two events. You can pretend to be asleep, hopefully resulting in your aunt giving up on you and putting the phone on her end back on the hook. This will result in the FSM going back to the `idle` state (and you going back to sleep).

If instead of ignoring it, you take your phone off the hook, it would stop ringing and the FSM would move to the `connected` state, leaving you to talk to your heart's content. When you are done with the call and hang up, the state reverts to `idle`.

[2] Or any other relative of your choice who tends to call you very early on a Saturday morning.

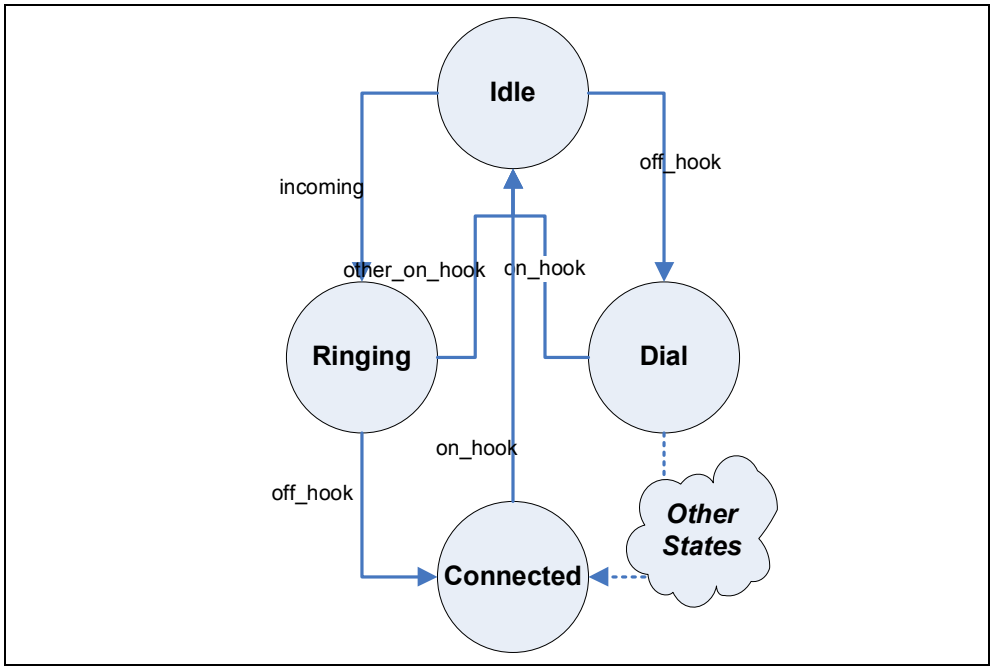

Figure 5-6. Fixed-line phone finite state machine

If the phone is in the `idle` state and you take it off the hook, a dial tone is started. Once the dial tone has started, the FSM changes to the `dial` state and you enter your aunt's phone number. Either you can hang up and your FSM goes back to the `idle` state, or your aunt picks up and you go to the `connected` state.

State machines are very common in all sorts of processing applications. In telecom systems, they are used not only to handle the state of equipment, as in the preceding example, but also in complex protocol stacks. The fact that Erlang handles them graciously is not a surprise. When prototyping with the early versions of Erlang between 1987 and 1991, it was the Plain Old Telephony System (POTS) finite state machines described in this section that the development team used to test their ideas of what Erlang should look like.

With a tail-recursive function for every state, actions implemented as function calls, and events represented as messages, this is what the code for the `idle` state would look like:

```
idle() ->
  receive
    {Number, incoming} ->
      start_ringing(),
      ringing(Number);
    off_hook ->
      start_tone(),
      dial()
  end.

ringing(Number) ->
  receive
    {Number, other_on_hook} ->
      stop_ringing(),
```

```
      idle();
    {Number, off_hook} ->
      stop_ringing(),
      connected(Number)
  end.

start_ringing() -> …
start_tone()    -> …
stop_ringing()  -> …
```

We leave the coding of the functions for the other states as an exercise.

A Mutex Semaphone

Let's look at another example of a finite state machine, this time implementing a mutex semaphore. A *semaphore* is a process which serializes access to a particular resource, guaranteeing mutual exclusion. Mutex semaphores might not be the first thing that comes to mind when working with Erlang, as they are commonly used in languages with shared memory. However, they can be used as a general mechanism for managing resources, not just memory.

Assume that only one process at a time is allowed to use the file server, thus guaranteeing that no two processes are simultaneously reading or writing to the same file. Before making any calls to the file server, the process wanting to access the file calls the `mutex:wait()` function, putting a lock on the server. When the process has finished handling the files, it calls the function `mutex:signal()`, removing the lock (see Figure 5-7).

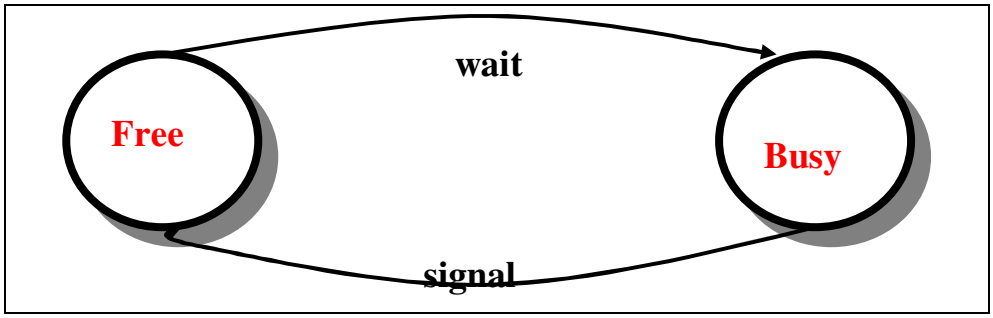

Figure 5-7. The mutex semaphore state diagram

If a process called PidB tries to call `mutex:wait()` when the semaphore is busy with PidA, PidB is suspended in its `receive` clause until PidA calls `signal/0`. The semaphore becomes available, and the process whose wait message is first in the message queue, PidB in our case, will be allowed to access the file server. The message sequence diagram in Figure 5-8 demonstrates this.

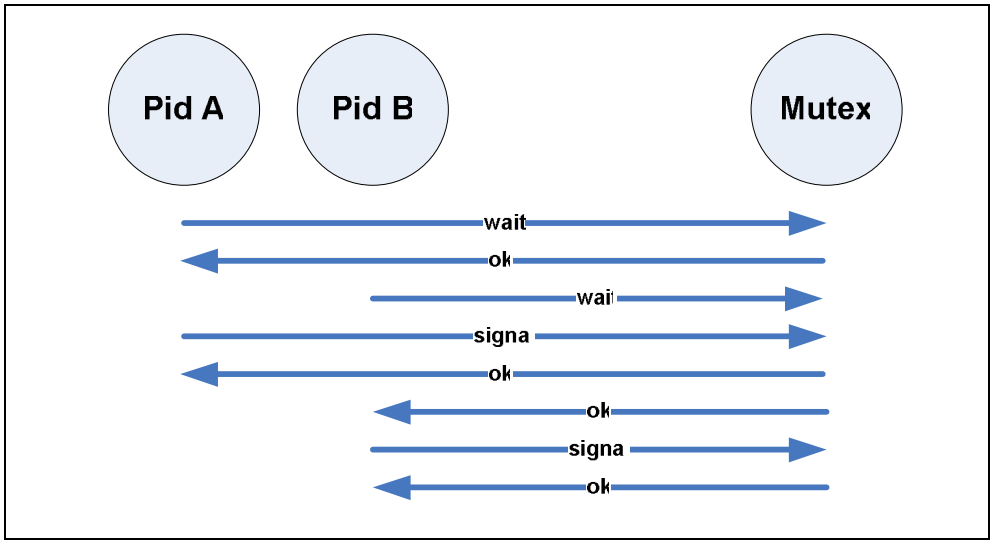

Figure 5-8. The mutex message sequence diagram

Look at the following code to get a feel for how to use tail-recursive functions to denote the states, and messages to denote events. And before reading on, try to figure out what the `terminate` function should do to clean up when the mutex is terminated.

```
-module(mutex).
-export([start/0, stop/0]).
-export([wait/0, signal/0]).
-export([init/0]).

start() ->
  register(mutex, spawn(?MODULE, init, [])).

stop() ->
  mutex ! stop.

wait() ->
  mutex ! {wait, self()},
  receive ok -> ok end.

signal() ->
  mutex ! {signal, self()}, ok.

init() ->
  free().

free() ->
  receive
    {wait, Pid} ->
      Pid ! ok,
      busy(Pid);
    stop ->
      terminate()
  end.

busy(Pid) ->
```

```
  receive
    {signal, Pid} ->
      free()
  end.

terminate() ->
  receive
    {wait, Pid} ->
      exit(Pid, kill),
      terminate()
  after
    0 -> ok
  end.
```

The `stop/0` function sends a stop message which is handled only in the `free` state. Prior to terminating the mutex process, all processes that are waiting for or holding the semaphore are allowed to complete their tasks. However, any process that attempts to wait for the semaphore after `stop/0` is called will be killed unconditionally in the `terminate/0` function.

Event Managers and Handlers

Try to picture a process which receives trace events generated in your system. You might want to do many things with these trace events, but you might not necessarily want to do all of them at the same time. You probably want to log all the trace events to file. If you are in front of the console, you might want to print them to standard I/O. You might be interested in statistics to determine how often certain errors occur, or if the event requires some action to be taken, you might want to send an SMS or SNMP[3] trap.

At any one time, you will want to execute some, if not all of these actions, and toggle between them. But if you walk away from your desk, you might want to turn the logging to the console off while maintaining the gathering of statistics and logging to file.

An *event manager* does what we just described. It is a process which receives a specific type of event and executes a set of actions determined by the type of event. These actions can be added and removed dynamically throughout the lifetime of the process, and are not necessarily defined or known when the code implementing the process is first written. They are collected in a module we call the *event handler*.

Large systems usually have an event handler for every type of event. Event types commonly include alarms, equipment state changes, errors, and trace events, just to mention a few. When they are received, one or more actions are applied to each event.

The most common form of event manager found in almost all industrial-grade systems handles alarms (see Figure 5-9). Alarms are raised when a problem occurs and are cleared when it goes away. They might require automated or manual intervention, but this is not always the case. An alarm would be raised if the data link between two devices

[3] SNMP stands for Simple Network Management Protocol. It is a standard used for controlling and monitoring systems over IP-based networks.

is lost and cleared if it recovers. Other examples include a cabinet door being opened, a fan breaking, or a TCP/IP connection being lost.

The alarm handler will often log these alarms, collect statistics, and filter and forward them to agents. Agents might receive the events and try to resolve the issues themselves. If a communication link is down, for example, an agent would automatically try to reconfigure the system to use the standby link, requesting human intervention only if the standby link goes down as well.

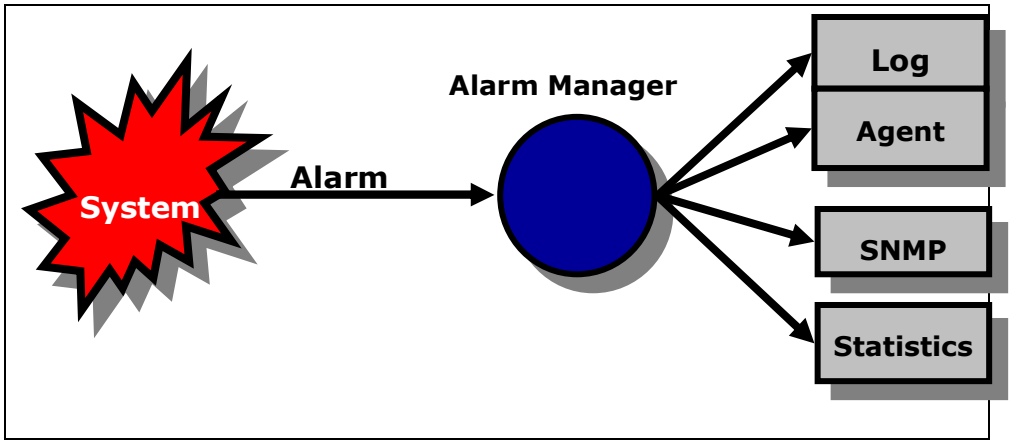

Figure 5-9. An alarm manager implemented as an event handler

A Generic Event Manager Example

Here is an example of an event manager which allows you to add and remove handlers during runtime. The code is completely generic and independent of the individual handlers. Handlers can be implemented in separate modules and have to export a number of functions, referred to as *callback functions*. These functions can be called by the event manager. We will cover them in a minute. Let's first look at how we've implemented the event manager, starting with its client functions:

`start(Name, HandlerList)`

> Will start a generic event manager, registering it with the alias `Name`. `HandlerList` is a list of tuples of the form `{Handler, Data}`, where `Handler` is the name of the handler callback module and `Data` is the argument passed to the handler's `init` callback function. `HandlerList` can be empty at startup, as handlers can be subsequently added using the `add_handler/2` call.

`stop(Name)`

> Will terminate all the handlers and stop the event manager process. It will return a list of items of the form `{Handler, Data}`, where `Data` is the return value of the `terminate` callback function of the individual handlers.

`add_handler(Name, Handler, Data)`

> Will add the handler defined in the callback module `Handler`, passing `Data` as an argument to the handler's `init` callback function.

`delete_handler(Name, Handler)`

Will remove the handler defined in the callback module `Handler`. The handler's `terminate` callback function will be called, and its return value will be the return value of this call. This call returns the tuple `{error, instance}` if `Handler` does not exist.

`get_data(Name, Handler)`

Will return the contents of the state variable of the `Handler`. This call returns the tuple `{error, instance}` if `Handler` does not exist.

`send_event(Name, Event)`

Will forward the contents of `Event` to all the handlers.

Here is the code for the generic event manager module:

```
-module(event_manager).
-export([start/2, stop/1]).
-export([add_handler/3, delete_handler/2, get_data/2, send_event/2]).
-export([init/1]).

start(Name, HandlerList) ->
  register(Name, spawn(generic_handler, init, [HandlerList])), ok.

init(HandlerList) ->
  loop(initialize(HandlerList)).

initialize([]) -> [];
initialize([{Handler, InitData}|Rest]) ->
  [{Handler, Handler:init(InitData)}|initialize(Rest)].
```

Here is an explanation of what the code is doing:

- The `start(Name, HandlerList)` function spawns the event manager process and registers it with the alias `Name`.
- The newly spawned process starts executing in the `init/1` function with a `HandlerList` tuple list of the format `{Handler, Data}` as an argument.
- We traverse the list in the `initialize/1` function calling `Handler:init(Data)` for every entry.
- The result of this call is stored in a list of the format `{Handler, State}`, where `State` is the return value of the `init` function.
- This list is passed as an argument to the event manager's `loop/1` function.

When stopping the event manager process, we send a `stop` message it receives in the `loop/1` function. If you are looking for `loop/1`, you will find it with the generic code at the end of this module. Receiving the `stop` message results in `terminate/1` traversing the list of handlers and calling `Handler:terminate(Data)` for every entry. The return value of these calls, a list of the format `{Handler, Value}`, is sent back to the process which originally called `stop/1` and becomes the return value of this function.

```
stop(Name) ->
  Name ! {stop, self()},
```

```
  receive {reply, Reply} -> Reply end.

terminate([]) -> [];
terminate([{Handler, Data}|Rest]) ->
  [{Handler, Handler:terminate(Data)}|terminate(Rest)].
```

Now we'll look at the client functions used to add, remove, and inspect the event handlers, as well as forwarding them the events. Through the `call/2` function, they send the request to the event manager process which handles them in `handle_msg/2`. Pay particular attention to the `send_event/2` call, which traverses the list of handlers, calling the callback function `Handler:handle_event(Event, Data)`. The return value of this call replaces the old `Data` and is used by the handler the next time one of its callbacks is invoked.

```
add_handler(Name, Handler, InitData) ->
  call(Name, {add_handler, Handler, InitData}).

delete_handler(Name, Hander) ->
  call(Name, {delete_handler, Hander}).

get_data(Name, Hander) ->
  call(Name, {get_data, Hander}).

send_event(Name, Event) ->
  call(Name, {send_event, Event}).

handle_msg({add_handler, Hander, InitData}, LoopData) ->
  {ok, [{Hander, Hander:init(InitData)}|LoopData]};

handle_msg({delete_handler, Hander}, LoopData) ->
  case lists:keysearch(Hander, 1, LoopData) of
    false ->
      {{error, instance}, LoopData};
    {value, {Hander, Data}} ->
      Reply = {data, Hander:terminate(Data)},
      NewLoopData = lists:keydelete(Hander, 1, LoopData),
      {Reply, NewLoopData}
  end;

handle_msg({get_data, Hander}, LoopData) ->
  case lists:keysearch(Hander, 1, LoopData) of
    false                    -> {{error, instance}, LoopData};
    {value, {Hander, Data}} -> {{data, Data}, LoopData}
  end;

handle_msg({send_event, Event}, LoopData) ->
  {ok, event(Event, LoopData)}.

event(_Event, []) -> [];
event(Event, [{Hander, Data}|Rest]) ->
  [{Hander, Hander:handle_event(Event, Data)}|event(Event, Rest)].
```

The following code, together with the `start` and `stop` functions we already covered, is a direct rip off from the process pattern example. By now, you should have spotted the reoccurring theme: *processes that handle very different tasks do so in similar ways, following a pattern.*

```
call(Name, Msg) ->
  Name ! {request, self(), Msg},
  receive {reply, Reply} -> Reply end.

reply(To, Msg) ->
  To ! {reply, Msg}.

loop(State) ->
  receive
    {request, From, Msg} ->
      {Reply, NewState} = handle_msg(Msg, State),
      reply(From, Reply),
      loop(NewState);
    {stop, From}  ->
      reply(From, terminate(State))
  end.
```

Event Handlers

In our event manager implementation, our event handlers have to export the following three callback functions:

`init(InitData)`

> Initializes the handler and returns a value which is used the next time a callback function belonging to the handler is invoked.

`terminate(Data)`

> Allows the handler to clean up. If we have opened files or sockets in the `init/1` callback, they would be closed here. The return value of `terminate/1` is passed back to the functions which originally instigated the removal of the handler. In our event manager example, they are the `delete_handler/2` and `stop/1` calls.

`handle_event(Event, Data)`

> Is called when an event is forwarded to the event manager through the `send_event/2` call. Its return value will be used the next time a callback function for this handler is invoked.

Using these callback functions, let's write two handlers—one which pretty-prints the events to the shell, and one which logs the events to file.

The `io_handler` event handler filters out events of the format `{raise_alarm, Id, Type}` and `{clear_alarm, Id, Type}`. All other events are ignored. In the `init/1` function, we set a counter which is incremented every time an event is handled.

The `handle_event/2` callback uses this counter every time an alarm event is received, displaying it together with information on the alarm.

```
-module(io_handler).
-export([init/1, terminate/1, handle_event/2]).

init(Count) -> Count.

terminate(Count) -> {count, Count}.

handle_event({raise_alarm, Id, Alarm}, Count) ->
```

```
  print(alarm, Id, Alarm, Count),
  Count+1;
handle_event({clear_alarm, Id, Alarm}, Count) ->
  print(clear, Id, Alarm, Count),
  Count + 1;
handle_event(Event, Count) ->
  Count.

print(Type, Id, Alarm, Count) ->
  Date = fmt(date()), Time = fmt(time()),
  io:format("#~w,~s,~s,~w,~w,~p~n",
            [Count, Date, Time, Type, Id, Alarm]).

fmt({AInt,BInt,CInt}) ->
  AStr = pad(integer_to_list(AInt)),
  BStr = pad(integer_to_list(BInt)),
  CStr = pad(integer_to_list(CInt)),
  [AStr,$:,BStr,$:,CStr].

pad([M1])  -> [$0,M1];
pad(Other) -> Other.
```

The second handler that we implement logs all the events of the format `{EventType, Id, Description}` in a comma-separated file, ignoring everything else which is not a tuple of size 3.

We open the file in the `init/1` function, write to it in `handle_event/2`, and close it in the `terminate` function. As this file will probably be read and manipulated by other programs, we will provide more detail in the information we write to it and spend less effort with its formatting. Instead of `time()` and `date()`, we use the `now()` BIF which gives us a timestamp with a much higher level of accuracy. It returns a tuple containing the mega seconds, seconds, and microseconds that have elapsed since January 1, 1970. When the `log_handler` is deleted from the event manager, the `terminate/2` call will close the file.

```
-module(log_handler).

-export([init/1, terminate/1, handle_event/2]).

init(File) ->
  {ok, Fd} = file:open(File, write),
  Fd.

terminate(Fd) -> file:close(Fd).

handle_event({Action, Id, Event}, Fd) ->
  {MegaSec, Sec, MicroSec} = now(),
  Args = io:format(Fd, "~w,~w,~w,~w,~w,~p~n",
                   [MegaSec, Sec, MicroSec, Action, Id, Event]),
  Fd;
handle_event(_, Fd) ->
  Fd.
```

Try out the event manager and the two handlers we've implemented in the shell. We start the event manager with the `log_handler`, after which we add and delete the

io_handler. In between, we generate a few alarms and test the other client functions we've implemented in the event manager work.

```
1> event_manager:start(alarm, [{log_handler, "AlarmLog"}]).
ok
2> event_manager:send_event(alarm, {raise_alarm, 10, cabinet_open}).
ok
3> event_manager:add_handler(alarm, io_handler, 1).
ok
4> event_manager:send_event(alarm, {clear_alarm, 10, cabinet_open}).
#1,2009:03:16,08:33:14,clear,10,cabinet_open
ok
5> event_manager:send_event(alarm, {event, 156, link_up}).
ok
6> event_manager:get_data(alarm, io_handler).
{data,2}
7> event_manager:delete_handler(alarm, stats_handler).
{error,instance}
8> event_manager:stop(alarm).
[{io_handler,{count,2}},{log_handler,ok}]
```

Exercises

Exercise 5-1: A Database Server

Write a database server that stores a database in its loop data. You should register the server and access its services through a functional interface. Exported functions in the my_db.erl module should include:

```
my_db:start() ⇒ ok.
my_db:stop() ⇒ ok.
my_db:write(Key, Element) ⇒ ok.
my_db:delete(Key) ⇒ ok.
my_db:read(Key) ⇒ {ok, Element} | {error, instance}.
my_db:match(Element) ⇒ [Key1, ..., KeyN].
```

Hint: Use the db.erl module as a backend and use the server skeleton from the echo server in Exercise 4-1.

Example:

```
1> my_db:start().
ok
2> my_db:write(foo, bar).
ok
3> my_db:read(baz).
{error, instance}
4> my_db:read(foo).
{ok, bar}
5> my_db:match(bar).
[foo]
```

Exercise 5-2: Changing the Frequency Server

Using the frequency server example in this chapter, change the code to ensure that only the client who allocated a frequency is allowed to deallocate it. Make sure that deallocating a frequency which has not been allocated does not make the server crash.

Hint: Use the `self()` BIF in the `allocate` and `deallocate` functions called by the client.

Extend the frequency server so that it can be stopped only if no frequencies are allocated.

Finally, test your changes to see whether they still allow individual clients to allocate more than one frequency at a time. This was previously possible by calling `allocate_frequency/0` more than once. Limit the number of frequencies a client can allocate to three.

Exercise 5-3: Swapping Handlers

What happens if you want to close and open a new file in the `log_handler`? You would have to call `event_manager:delete_handler/2` immediately followed by `event_manager:add_handler/2`. The risk with this is that in between these two calls, you might miss an event. Therefore, implement the following function,

```
event_manager:swap_handlers(Name, OldHandler, NewHandler)
```

which swaps the handlers atomically, ensuring that no events are lost. To ensure that the state of the handlers is maintained, pass the return value of `OldHandler:terminate/1` to the `NewHandler:init/1` call.

Exercise 5-4: Event Statistics

Write a `stats_handler` module which takes the first and second elements of the event tuple `{Type, Id, Description}` in our example and keep a count of how many times the combination of `{Type, Description}` occurs. Users should be able to retrieve these statistics by using the client function `event_manager:get_data/2`.

Exercise 5-5: Phone FSM

Complete the coding of the phone FSM example, and then instrument it with logging using an event handler process. This should record enough information to enable billing for the use of the phone.

12

OTP Behaviors

In previous chapters, we introduced patterns that recur when you program using the Erlang concurrency model. We discussed functionality common to concurrent systems, and you saw that processes will handle very different tasks in a similar way. We also emphasized special cases and potential problems that have to be handled when dealing with concurrency.

For example, picture a project with 50 developers spread across several geographic locations. If the project is not properly coordinated, how many different client/server implementations might the project end up with? Even more dangerous, how many of these implementations will handle special borderline cases and concurrency-related errors correctly, if at all? Without a code review, can you be sure there is a uniform way across the system to handle server crashes that occur after clients have sent a request to the server? Or guarantee that the response from a request is indeed the response, and not just any message that conforms to the internal message protocol?

OTP behaviors address all of these issues by providing library modules that implement the most common concurrent design patterns. Behind the scenes, without the programmer having to be aware of it, the library modules ensure that errors and special cases are handled in a consistent way. As a result, OTP behaviors provide a set of standardized building blocks used in designing and building industrial-grade systems. The subject of OTP behaviors and their related middleware is vast. In this chapter, we provide the overview you need to get started.

Introduction to OTP Behaviors

OTP behaviors are a formalization of process design patterns. They are implemented in library modules that are provided with the standard Erlang distribution. These library modules do all of the generic process work and error handling. The specific code, written by the programmer, is placed in a separate module and called through a set of predefined callback functions.

OTP behaviors include *worker* processes, which do the actual processing, and *supervisors*, whose task is to monitor workers and other supervisors. Worker behaviors, often denoted in diagrams as circles, include servers, event handlers, and finite state machines. Supervisors, denoted in illustrations as squares, monitor their children, both workers and other supervisors, creating what is called a *supervision tree* .(see Figure 12-1)

Supervision trees are packaged into a behavior called an *application*. OTP applications not only are the building blocks of Erlang systems, but also are a way to package reusable components. Industrial-grade systems consist of a set of loosely coupled, possibly distributed, applications. These applications either are part of the standard Erlang distribution, or are specific applications developed by you, the programmer.

Do not confuse OTP applications with the more general concept of an application, which usually refers to a more complete system that solves a high-level task. Examples of OTP applications include the Mnesia database, which we cover in Chapter 13; an SNMP agent; or the mobile subscriber database introduced in Chapter 10, which we will convert to an application using behaviors later in this chapter. An OTP application as a reusable component that packages library modules together with supervisor and worker processes. From now on, when we refer to an application, we will mean an OTP application.

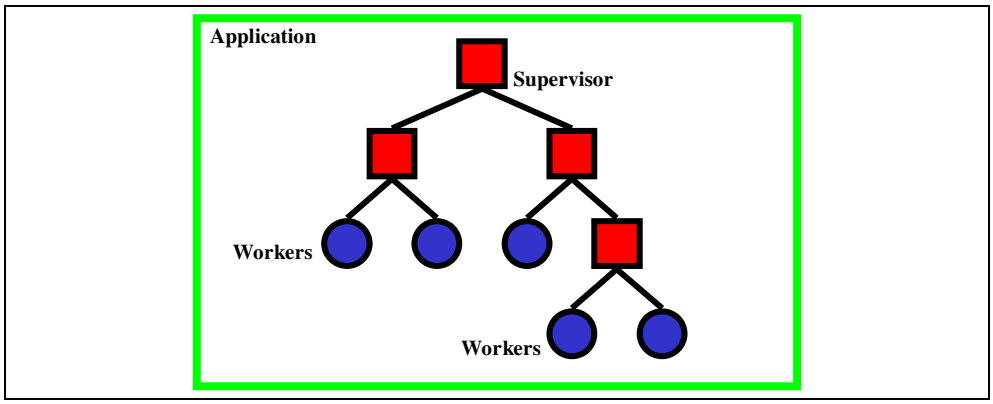

Figure 12-1. Supervision tree in an application

The behavior module contains all of the generic code. Although it is possible to implement your own behavior module, doing so is rare because the behavior modules that come as part of the Erlang/OTP distribution will cater for most of the design patterns you would use in your code. The generic functionality provided in a behavior module includes operations such as the following:

- Spawning and possibly registering the process
- Sending and receiving client messages as synchronous or asynchronous calls, including defining the internal message protocol
- Storing the loop data and managing the process loop
- Stopping the process

Although the behavior module is provided, the programmer has to develop the callback module (see Figure 12-2). We covered the concept of callback modules in Chapter 5. A callback module contains all of the specific code required to deliver the desired

functionality. The specific code is invoked through a callback interface that is standardized for each behavior.

The *loop data* is a variable that will contain the data the behavior needs to store in between calls. After the call, an updated variant of the loop data is returned. This updated loop data, often referred to as the *new loop data*, is passed as an argument in the next call. Loop data is also commonly referred to as *behavior state*.

The functionality to be included in the callback module to deliver the specific behavior required includes the following:

- Initializing the process loop data and, if the process is registered, the process name
- Handling the specific client requests and, if synchronous, the replies sent back to the client
- Handling and updating the process loop data in between the process requests
- Cleaning up the process loop data upon termination

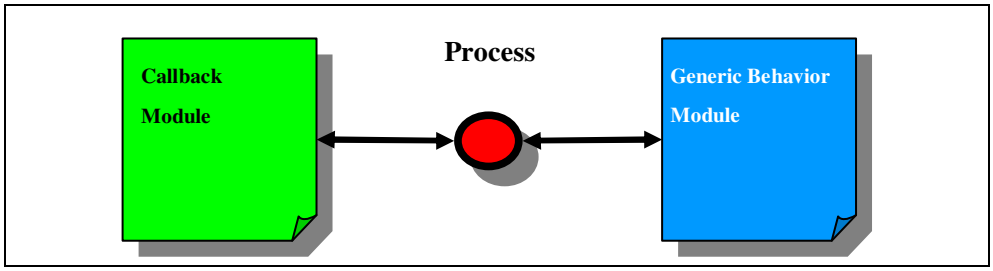

Figure 12-2. Splitting the code into generic and specific modules

There are many advantages to splitting the code into generic behavior libraries and specific callback modules:

- Because many of the special cases and errors that might occur are already handled in the solid, well-tested library, you can expect fewer bugs in your product.
- For this reason, and also because so much of the code is already written for you, you can expect to have a shorter time to market.
- It forces the programmer to write code in a way that avoids errors typically found in concurrent applications.
- Finally, your whole team will come to share a common programming style. When reading someone else's code while armed with a basic comprehension of the existing behaviors, no effort is required to understand the client/server protocol, looking for where and how processes are started or terminated or how the loop data is handled. All of it is managed by the generic behavior library. Instead of having to focus on how everything is done, you can focus on what is being done *specifically* in this case, as coded in the callback module.

In the sections that follow, we will look at some of the most important behaviors—including generic servers and supervisors—and how to package them together into applications.

Generic Servers

Generic servers that implement client/server behaviors are defined in the `gen_server` behavior that comes as part of the standard library application. In this chapter, you will use the mobile customer database example from Chapter 10 to understand how the callback principle works. If you do not remember the example, take a quick look at it before proceeding.

You will rewrite the `usr.erl` module, migrating it from an Erlang process to a `gen_server` behavior. In doing so, you will not touch the `usr_db` module, keeping the backend database as it is. When working your way through the example, if you are interested in the details, have the manual pages for the `gen_server` module at hand.

Starting Your Server

With the `gen_server` behavior, instead of using the `spawn` and `spawn_link` BIFs, you will use the `gen_server:start/4` and `gen_server:start_link/4` functions.

The main difference between `spawn` and `start` is the *synchronous* nature of the call. Using `start` instead of `spawn` makes starting the worker process more deterministic and prevents unforeseen race conditions, as the call will not return the pid of the worker until it has been initialized. You call the functions as follows (we show two variants for each of the two functions):

```
gen_server:start_link(ServerName, CallBackModule, Arguments, Options)
gen_server:start(ServerName, CallBackModule, Arguments, Options)

gen_server:start_link(CallBackModule, Arguments, Options)
gen_server:start(CallBackModule, Arguments, Options)
```

In the preceding calls:

`ServerName`

> Is a tuple of the format `{local, Name}` or `{global, Name}`, denoting a local or global `Name` for the process if it is to be registered. If you do not want to register the process and instead reference it using its pid, you omit the argument and use the `start_link/3` or `start/3` call instead.

`CallbackModule`

> Is the name of the module in which the specific callback functions are placed.

`Arguments`

> Is a valid Erlang term that is passed to the `init/1` callback function. You can choose what type of term to pass: if you have many arguments to pass, use a list or a tuple; if you have none, pass an atom or an empty list, ignoring it in the callback function.

`Options`

> Is a list that allows you to set the memory management flags `fullsweep_after` and `heapsize`, as well as tracing and debugging flags. Most behavior implementations just pass the empty list.

The `start` functions will spawn a new process that calls the `init(Arguments)` callback function in the `CallbackModule`, with the `Arguments` supplied. The `init` function must initialize the `LoopData` of the server and has to return a tuple of the format `{ok, LoopData}`. `LoopData` contains the first instance of the loop data that will be passed between the callback functions. If you want to store some of the arguments you passed to the `init` function, you would do so in the `LoopData` variable.

The obvious difference between the `start_link` and `start` functions is that `start_link` links to its parent and `start` doesn't. This needs a special mention, however, as it is an OTP behavior's responsibility to link itself to the supervisor. The `start` functions are often used when testing behaviors from the shell, as a typing error causing the shell process to crash would not affect the behavior. All of the `start` and `start_link` variants return `{ok, Pid}`.

Before going ahead with the example, let's quickly review what we have discussed so far. You start a `gen_server` behavior using the `gen_server:start_link` call. This results in a new process that calls the `init/1` callback function. This function initializes the `LoopData` and returns the tuple `{ok, LoopData}`.

In our example, we call `start_link/4`, registering the process with the same name as the callback module, calling the `MODULE` macro. We pass one argument, the filename of the Dets table. The options list is kept empty.

```
start_link(FileName) ->
  gen_server:start_link({local, ?MODULE}, ?MODULE, FileName, []).

init(FileName) ->
  usr_db:create_tables(FileName),
  usr_db:restore_backup(),
  {ok, null}.
```

Although the supervisor process might call the `start_link/4` function, the `init/1` callback is called by a different process: the one that was just spawned. We don't really need the `LoopData` variable in our server, as the ETS and Dets tables are named. Nonetheless, a value still has to be included when returning the `{ok, LoopData}` structure, so we'll get around it by returning the atom `null`. Had the ETS and Dets tables not been `named_tables`, we would have passed their references here.

Do only what is necessary and minimize the operations in your `init` function, as the call to `init` is a synchronous call which prevents all of the other serialized processes from starting until it terminates.

Passing Messages

If you want to send a message to your server, you use the following calls:

```
gen_server:cast(Name, Message)
gen_server:call(Name, Message)
```

In the preceding calls:

`Name`

> Is either the *local* registered name of the server or the tuple `{global, Name}`. It could also be the process identifier of the server.

`Message`

Is a valid Erlang term containing a message passed on to the server.

For *asynchronous* message requests, you use `cast/2`. If you're using a pid, the call will immediately return the atom `ok`, regardless of whether the `gen_server` to which you are sending the message is alive. These semantics are no different from the standard `Name ! Message` construct, where if the registered process `Name` does not exist, the calling process terminates.

Upon receiving the message, `gen_server` will call the callback function `handle_cast(Message, LoopData)` in the callback module. `Message` is the argument passed to the `cast/2` function, and `LoopData` is the argument originally returned by the `init/1` callback function. The `handle_cast/1` callback function handles the specifics of the message, and upon finishing, it has to return the tuple `{noreply, NewLoopData}`. In future calls to the server, the `NewLoopData` value most recently returned will be passed as an argument when a message is sent to the server.

If you want to send a *synchronous* message to the server, you use the `call/2` function. Upon receiving this message, the process uses the `handle_call(Message, From, LoopData)` function in the callback module. It contains specific code for the particular server, and having completed, it returns the tuple `{reply, Reply, NewLoopData}`. Only now does the `call/3` function synchronously return the value `Reply`. If the process you are sending a message to does not exist, regardless of whether it is registered, the process invoking the call function terminates.

Let's start by taking two functions from our service API; we will provide the whole program later. They are called by the client process and result in a synchronous message being sent to the server process registered with the same name as the callback module. Note how we are validating the data on the client side. If the client sends incorrect information, it terminates.

```
set_status(CustId, Status) when Status==enabled; Status==disabled->
     gen_server:call(?MODULE, {set_status, CustId, Status}).

delete_disabled() ->
     gen_server:call(?MODULE, delete_disabled).
```

Upon receiving the messages, the `gen_server` process calls the `handle_call/3` callback function dealing with the messages in the same order in which they were sent.

```
handle_call({set_status, CustId, Status}, _From, LoopData) ->
    Reply = case usr_db:lookup_id(CustId) of
      {ok, Usr} ->
        usr_db:update_usr(Usr#usr{status=Status});
      {error, instance} ->
        {error, instance}
    end,
    {reply, Reply, LoopData};

handle_call(delete_disabled, _From, LoopData) ->
    {reply, usr_db:delete_disabled(), LoopData}.
```

Note the return value of the callback function. The tuple contains the *control atom* `reply`, telling the `gen_server` generic code that the second element of the tuple is the

`Reply` to be sent back to the client. The third element of the tuple is the new `LoopData`, which, in a new iteration of the server, is passed as the third argument to the `handle_call/3` function; in both cases here it is unchanged. The argument `_From` is a tuple containing a unique message reference and the client process identifier. The tuple as a whole is used in library functions that we will not be discussing in this chapter. In the majority of cases, you will not need it.

The `gen_server` library module has a number of mechanisms and safeguards built in that function behind the scenes. If your client sends a synchronous message to your server and you do not get a response within five seconds, the process executing the `call/2` function is terminated. You can override this by using the following code,

```
gen_server:call(Name, Message, Timeout)
```

where `Timeout` is a value in milliseconds or the atom `infinity`. The timeout mechanism was originally put in place for deadlock prevention purposes, ensuring that servers that accidentally call each other are terminated after the default timeout. The crash report would be logged, and hopefully would result in a patch. Most applications will function appropriately with a timeout of five seconds, but under very heavy loads you might have to fine-tune the value and possibly even use `infinity`; this choice is very application-dependent. All of the critical code in Erlang/OTP uses `infinity`.

Other safeguards when using the `gen_server:call/2` function include the case of sending a message to a non-existing server or a server that crashes before sending its reply. In both cases, the calling process will terminate. In raw Erlang, sending a message that is never pattern-matched in a `receive` clause is a bug which can cause a memory leak.

What do you think happens if you do a call or a cast to your server, but do not handle the message in the `handle_call/3` and `handle_cast/2` calls, respectively? In OTP, when a call or a cast is called, the message will always be extracted from the process mailbox and the respective callback functions are invoked. If none of the callback functions pattern-match the message passed as the first argument, the process will crash with a function clause error. As a result, such issues will be caught in the early stages of the testing phase and dealt with accordingly.

Stopping the Server

How do you stop the server? In your `handle_call/3` and `handle_cast/2` callback functions, instead of returning `{reply, Reply, NewLoopData}` or `{noreply, NewLoopData}`, you can return `{stop, Reason, Reply, NewLoopData}` or `{stop, Reason, NewLoopData}`, respectively. Something has to trigger this return value, often a stop message sent to the server. Upon receiving the `stop` tuple containing the `Reason` and `LoopData`, the generic code executes the `terminate(Reason, LoopData)` callback.

The `terminate` function is the natural place to insert the code needed to clean up the `LoopData` of the server and any other persistent data used by the system. In this example, it would mean closing the ETS and Dets tables. The `stop` call does not have to occur within a synchronous call, so let's use `cast` when implementing it:

```
stop() ->
  gen_server:cast(?MODULE, stop).
```

```
handle_cast(stop, LoopData) ->
  {stop, normal, LoopData}.

terminate(_Reason, _LoopData) ->
  usr_db:close_tables().
```

Remember that `stop/0` will be called by the client process, while the `handle_cast/2` and `handle_call/2` are called by the behaviour process. In the `handle_cast/2` callback, we return the reason `normal` in the `stop` construct. Any reason other than `normal` will result in an error report being generated.

With thousands of generic servers potentially being spawned and terminated every second, generating error reports for every one of them is not the way to go. You should return a *non-normal* value only if something that should not have happened occurs and you have no way to recover. A socket being closed or a corrupt message from an external port is not a reason to generate a non-normal termination. On the other hand, corrupt internal data or a missing configuration file is.

If your server crashes because of a runtime error, `terminate/2` will be called. But if your behavior receives an `EXIT` signal from its parent, `terminate` will be called only if you are trapping exits. Watch out for this special case, as we've been caught by it many times, especially when starting the behavior from the shell using `start_link`.

> Use of the behavior callbacks as library functions and invoking them from other parts of your program is an extremely bad practice. For example, you should never call `usr_db:init(FileName)` from another module to create and populate your database. Calls to behavior callback functions should originate only from the behavior library modules as a result of an event occurring in the system, and *never* directly by the user.

The Example in Full

Here is the `usr.erl` module from Chapter 10, rewritten as a `gen_server` behavior:

```
%%% File    : usr.erl
%%% Description : API and gen_server code for cellphone user db

-export([start_link/0, start_link/1, stop/0]).
-export([init/1, terminate/2, handle_call/3, handle_cast/2]).
-export([add_usr/3, delete_usr/1, set_service/3, set_status/2,
         delete_disabled/0, lookup_id/1]).
-export([lookup_msisdn/1, service_flag/2]).
-behavior(gen_server).

-include("usr.hrl").

%% Exported Client Functions
%% Operation & Maintenance API

start_link() ->
  start_link("usrDb").

start_link(FileName) ->
```

```
  gen_server:start_link({local, ?MODULE}, ?MODULE, FileName, []).

stop() ->
  gen_server:cast(?MODULE, stop).

%% Customer Services API

add_usr(PhoneNum, CustId, Plan) when Plan==prepay; Plan==postpay ->
  gen_server:call(?MODULE, {add_usr, PhoneNum, CustId, Plan}).

delete_usr(CustId) ->
  gen_server:call(?MODULE, {delete_usr, CustId}).

set_service(CustId, Service, Flag) when Flag==true; Flag==false ->
  gen_server:call(?MODULE, {set_service, CustId, Service, Flag}).

set_status(CustId, Status) when Status==enabled; Status==disabled->
  gen_server:call(?MODULE, {set_status, CustId, Status}).

delete_disabled() ->
  gen_server:call(?MODULE, delete_disabled).

lookup_id(CustId) ->
  usr_db:lookup_id(CustId).

%% Service API

lookup_msisdn(PhoneNo) ->
  usr_db:lookup_msisdn(PhoneNo).

service_flag(PhoneNo, Service) ->
  case usr_db:lookup_msisdn(PhoneNo) of
    {ok,#usr{services=Services, status=enabled}} ->
      lists:member(Service, Services);
    {ok, #usr{status=disabled}} ->
      {error, disabled};
    {error, Reason} ->
      {error, Reason}
  end.

%% Callback Functions

init(FileName) ->
  usr_db:create_tables(FileName),
  usr_db:restore_backup(),
  {ok, null}.

terminate(_Reason, _LoopData) ->
  usr_db:close_tables().

handle_cast(stop, LoopData) ->
  {stop, normal, LoopData}.

handle_call({add_usr, PhoneNo, CustId, Plan}, _From, LoopData) ->
  Reply = usr_db:add_usr(#usr{msisdn=PhoneNo,
                              id=CustId,
                              plan=Plan}),
```

```
  {reply, Reply, LoopData};

handle_call({delete_usr, CustId}, _From, LoopData) ->
  Reply = usr_db:delete_usr(CustId),
  {reply, Reply, LoopData};

handle_call({set_service, CustId, Service, Flag}, _From, LoopData) ->
  Reply = case usr_db:lookup_id(CustId) of
    {ok, Usr} ->
      Services = lists:delete(Service, Usr#usr.services),
      NewServices = case Flag of
        true  -> [Service|Services];
        false -> Services
      end,
      usr_db:update_usr(Usr#usr{services=NewServices});
    {error, instance} ->
      {error, instance}
  end,
  {reply, Reply, LoopData};

handle_call({set_status, CustId, Status}, _From, LoopData) ->
  Reply = case usr_db:lookup_id(CustId) of
    {ok, Usr} ->
      usr_db:update_usr(Usr#usr{status=Status});
    {error, instance} ->
      {error, instance}
  end,
  {reply, Reply, LoopData};

handle_call(delete_disabled, _From, LoopData) ->
  {reply, usr_db:delete_disabled(), LoopData}.
```

Running gen_server

When testing the `gen_server` instance in the shell, you get exactly the same behavior as when you used the server process that you coded yourself. However, the code is more solid, as deadlocks, server crashes, timeouts, and other errors related to concurrent programming are handled behind the scenes.

```
1> c(usr).
/Users/Francesco/otp/usr.erl:11: Warning: undefined callback function code_change/3
(behaviour 'gen_server')
/Users/Francesco/otp/usr.erl:11: Warning: undefined callback function handle_info/2
(behaviour 'gen_server')
{ok,usr_db}
2> c(usr_db).
{ok,usr_db}
3> rr("usr.hrl").
[usr]
4> usr:start_link().
{ok,<0.86.0>}
5> usr:add_usr(700000000, 0, prepay).
ok
6> usr:set_service(0, data, true).
ok
7> usr:lookup_id(0).
{ok,#usr{msisdn = 700000000,id = 0,status = enabled,
```

```
        plan = prepay,
        services = [data]}}
8> usr:set_status(0, disabled).
ok
9> usr:service_flag(700000000,lbs).
{error,disabled}
10> usr:stop().
ok
```

Did you notice the `-behavior(gen_server)` directive in the module? This tells the compiler that your module is a `gen_server` callback module, and as a result, it has to expect a number of callback functions. If all callback functions are not implemented, you will get the warnings you noticed as a result of the `compile` operation in the first command line. Don't write your code to avoid these warnings. If your server has no asynchronous calls, you will obviously not need a `handle_cast/2`. Ignore the warnings.

> British or Canadian readers: don't despair or shake your heads! You are welcome to use the English (U.K.) spelling in your directive: `-behaviour(gen_server)`. The compiler is bilingual and can handle both U.S. and U.K. English.

What happens if you send a message to the server using raw Erlang message passing of the form `Pid!Msg`? It should be possible, as the `gen_server` is an Erlang process capable of sending and receiving messages like any other process. Don't be shy; try it:

```
11> {ok, Pid} = usr:start_link().
{ok,<0.119.0>}
12> Pid ! hello.
hello

=ERROR REPORT==== 24-Jan-2009::18:08:07 ===
** Generic server usr terminating
** Last message in was hello
** When Server LoopData == null
** Reason for termination ==
** {'function not exported',[{usr,handle_info,[hello,null]},
                             {gen_server,handle_msg,5},
                             {proc_lib,init_p,5}]}
** exception exit: undef
     in function  usr:handle_info/2
        called as usr:handle_info(hello,null)
     in call from gen_server:handle_msg/5
     in call from proc_lib:init_p/5
```

Oops! Something did not go according to plan. Look at the error and try to figure out what happened. Use of `Pid!Msg` does not comply with the internal OTP message protocol. Upon receiving a message that is not compliant, the `gen_server` process tries to call the function `usr:handle_info(hello, null)`, where `hello` is the message and `null` is the loop data.

The callback function `handle_info/2`[1] is called whenever the process receives a message it doesn't recognize. These could include "node down" messages from nodes you are monitoring, exit signals from processes you are linked to, or simply messages sent using the `...!...` construct. If you are expecting such messages but are not interested in them, add the following definition to your callback module, and don't forget to export it:

```
handle_info(_Msg, LoopData) ->
  {noreply, LoopData}.
```

If, on the other hand, you do want to do something with the messages, you should pattern-match them in the first argument of the call. If your server is not expecting non-OTP-compliant messages, don't add the `handle_info/2` call, which ignores incoming messages, "just in case." Doing so is considered defensive programming, which will probably make any fault you are hiding hard to detect.

One of the downsides of OTP is the layering that the various behavior modules require. This will affect performance. In the attempt to save a few microseconds from their calls, developers have been known to use the `Pid ! Msg` construct instead of a `gen_server` cast, handling their messages in the `handle_info/2` callback.

Don't do this! You will make your code impossible to support and maintain, as well as losing many of the advantages of using OTP in the first place. If you are obsessed with saving microseconds, try to hold on and optimize only when you know your program is not fast enough. We discuss optimizations in Chapter 20 and will cover there what really affects the performance of your code.

Before we look at the next behavior, here is a summary of the exported `gen_server` API, the resulting callback functions, and their expected return values:

Setup

The following calls,

```
start(Name, Mod, Arguments, Opts)
start_link(Name, Mod, Arguments, Opts),
```

where `Name` is an optional argument, spawn a new process. The process will result in the callback function `init(Arguments)` being called, which should return one of the values `{ok, LoopData}` or `{stop, Reason}`. If `init/1` returns `{stop, Reason}` the `terminate/2` "cleanup" function will *not* be called.

Synchronous communication

Use `call(Name, Msg)` to send a synchronous message to your server. It will result in the callback function `handle_call(Msg, From, LoopData)` being called by the server process. The expected return values include the following:

```
{reply, Reply, NewLoopData}
{stop, Reason, Reply, NewLoopData}.
```

[1] Did you notice it in the compiler warning in the example?

Asynchronous communication

If you want to send an asynchronous message, use `cast(Name, Msg)`. It will be handled in the `handle_cast(Msg, LoopData)` callback function, returning either `{noreply, NewLoopData}` or `{stop, Reason, NewLoopData}`.

Non-OTP-compliant messages

Upon receiving non-OTP-compliant messages, `gen_server` will execute the `handle_info(Msg, LoopData)` callback function. The function should return either `{noreply, NewLoopData}` or `{stop, Reason, NewLoopData}`.

Termination

Upon receiving a `stop` construct from one of the callback functions (except for `init`), or upon abnormal process termination when trapping exits, the `terminate(Reason, LoopData)` callback is invoked. In `terminate/2`, you would typically undo things you did in `init/1`. Its return value is ignored.

Supervisors

The supervisor behavior's task is to monitor its children and, based on some preconfigured rules, take action when they terminate. The children that make up the supervision tree include both supervisors and worker processes. Worker processes are OTP behaviors including `gen_server`, `gen_fsm` (supporting finite state machine behavior), and `gen_event` (which provides event-handling functionality).

Worker processes have to link themselves to the supervisor behavior and handle specific system messages that are not exposed to the programmer. This is different from the way in which one process links to another in raw Erlang, and we cannot mix the two mechanisms. For this reason, it is not possible to add Erlang processes to the supervision tree in the form you know them. So for the remainder of this section, we will stick to describing supervision within the OTP framework.

You start a supervisor using the `start` or `start_link` function:

```
supervisor:start_link(ServerName, CallBackModule, Arguments)
supervisor:start(ServerName, CallBackModule, Arguments)
supervisor:start_link(CallBackModule, Arguments)
supervisor:start(CallBackModule, Arguments)
```

In the preceding calls:

`ServerName`

Is the *name* to be registered for the supervisor, and is a tuple of the format `{local, Name}` or `{global, Name}`. If you do not want to register the supervisor, you use the functions of arity two.

`CallbackModule`

Is the name of the module in which the `init/1` callback function is placed.

`Arguments`

Is a valid Erlang term that is passed to the `init/1` callback function when it is called.

Note that the supervisor, unlike the `gen_server`, does not take any options. The `start` and `start_link` functions will spawn a new process that calls the `init/1` callback function. Upon initializing the supervisor, the `init` function has to return a tuple of the following format:

```
{ok, {SupervisorSpecification, ChildSpecificationList}}
```

The *supervisor specification* is a tuple containing information on how to handle process crashes and restarts. The *child specification* list specifies which children the supervisor has to start and monitor, together with information on how to terminate and restart them.

Supervisor Specifications

The supervisor specification is a tuple consisting of three elements describing how the supervisor should react when a child terminates:

```
{RestartStrategy, AllowedRestarts, MaxSeconds}
```

The restart strategy determines how other children are affected if one of their siblings terminates. It can be one of the following:

`one_for_one`

Will restart the child that has terminated, without affecting any of the other children. You should pick this strategy if all of the processes at this level of the supervision tree are not dependent on each other.

`one_for_all`

Will terminate all of the children and restart them. You should use this if there is a strong dependency among all of the children regardless of the order in which they were started.

`rest_for_one`

Will terminate all of the children that were started after the child that crashed, and will restart them. This strategy assumes that processes are started in order of dependency, where spawned processes are dependent only on their already started siblings.

What will happen if your process gets into a cyclic restart? It crashes and is restarted, only to come across the same corrupted data, and as a result, it crashes again. This can't go on forever! This is where `AllowedRestarts` comes in, by specifying the maximum number of abnormal terminations the supervisor is allowed to handle in `MaxSeconds` seconds. If more abnormal terminations occur than are allowed, it is assumed that the supervisor has not been able to resolve the problem, and terminates. The supervisor's supervisor receives the exit signal and, based on its configuration, decides how to proceed.

Finding reasonable values for `AllowedRestarts` and `MaxSeconds` is not easy, as they will be application-dependent. In production, we've used anything from one restart per second to one per hour. Your choice will have to depend on what your child

processes do, how many of them you expect the supervisor to monitor, and how you've set up your supervision strategy.

Child Specifications

The second argument in the structure returned by the `init/1` function is a list of child specifications. Child specifications provide the supervisor with the properties of each of its children, including instructions on how to start it. Each child specification is of the following form:

```
{Id, {Module, Function, Arguments}, Restart, Shutdown, Type, ModuleList}
```

In the preceding code:

`Id`

Is a unique identifier for a particular child within a supervisor. As a child process can crash and be restarted, its process identifier might change. The identifier is used instead.

The supervisor uses the tuple `{Module, Function, Arguments}` to start the child process. The supervisor has to eventually call the `start_link` function for the particular OTP behavior, and return `{ok, Pid}`.

`Restart`

Is one of the atoms `transient`, `temporary`, or `permanent`. Transient processes are never restarted. Temporary processes are restarted only if they terminate abnormally, and permanent processes are always restarted, regardless of whether the termination was normal or non-normal.

`Shutdown`

Specifies how many *milliseconds* a behavior trapping exits is allowed to execute in its `terminate` callback function after receiving the `shutdown` signal from its supervisor, either because the supervisor has reached its maximum number of allowed child restarts or because of a `rest_for_one` or `one_for_all` restart strategy.

If the child process has not terminated by this time, the supervisor will kill it unconditionally. `Shutdown` will also take the atom `infinity`, a value which should always be chosen if the process is a supervisor, or the atom `brutal_kill`, if the process is to be killed unconditionally.

`Type`

Specifies whether the child process is a `worker` or a `supervisor`.

`ModuleList`

Is a list of the modules that implement the process. The release handler uses it to determine which processes it should suspend during a software upgrade. As a rule of thumb, always include the behavior callback module.

In some cases, child specifications are created dynamically from a *config* file. In most cases, however, they are statically coded in the supervisor callback module. The `init/1` function is the only callback function that needs to be exported.

It can be easy to insert syntactical and semantic errors in child specification lists, as they tend to get fairly complex. The help function `check_childspecs/1` in the supervisor module takes a list of child specifications and returns `ok` or the tuple `{error, Reason}`. An example of a child specification for the mobile subscriber database will follow in the next section. To ensure that you understand what is happening, map all of the entries to their respective fields in the child specification structure.

Supervisor Example

In this example, the `usr_sup` module is a supervisor behavior, supervising one child that is the `usr` example of a `gen_server` from earlier in the chapter.

We'll start the supervisor using the `start_link/0` call. Note that we've omitted the option of passing a filename for the Dets tables, as it was originally included for test purposes. Pay particular attention to the child and the supervisor specifications returned by the `init/1` function.

```
-module(usr_sup).
-behavior(supervisor).

-export([start_link/0]).
-export([init/1]).

start_link() ->
  supervisor:start_link({local, ?MODULE}, ?MODULE, []).

init(FileName) ->
  UsrChild = {usr,{usr, start_link, []},
              permanent, 2000, worker, [usr, usr_db]},
  {ok,{{one_for_all,1,1}, [UsrChild]}}.
```

Now you can try it out from the shell. Do not test only positive cases; also try to kill the child and ensure that it has been restarted. Finally, kill the server more than `MaxRestart` times in `MaxSeconds` (twice in one second in this example), to see whether the supervisor terminates.

```
13> c(usr_sup).
{ok,usr_sup}
14> usr_sup:start_link().
{ok,<0.149.0>}
15> whereis(usr).
<0.150.0>
16> exit(whereis(usr), kill).
true
17> whereis(usr).
<0.156.0>
18> usr:lookup_id(0).
{ok,#usr{msisdn = 700000000,id = 0,status = disabled,
         plan = prepay,
         services = [data]}}
19> exit(whereis(usr), kill).
true
20> exit(whereis(usr), kill).
** exception exit: shutdown
```

When a process terminates, all the ETS tables that it created are destroyed. If you want ETS tables to survive process restarts without incurring the overhead of dealing with Dets tables or the filesystem, a trick is to let your supervisor create the tables in its `init/1` function, rather than in the processes spawned.

Dynamic Children

So far, we have looked only at static children. What if you need a supervisor that dynamically creates a child whose task is to handle a specific event, take care of the task, and terminate when completed? It could be for every incoming instant message (IM) or buddy update coming into your IM server. You can't specify these children in your `init` callback function, as they are created dynamically. Instead, you need to use the calls to functions `supervisor:???_child/2`:

```
supervisor:start_child(SupervisorName, ChildSpec)
supervisor:terminate_child(SupervisorName, Id)
supervisor:restart_child(SupervisorName, Id)
supervisor:delete_child(SupervisorName, Id).
```

In the preceding calls:

`SupervisorName`

Is either the process identifier of the supervisor or its registered name

`ChildSpec`

Is a single child specification tuple, as described in the section "Child Specifications"

`Id`

Is the unique child identifier defined in the `ChildSpec`

Of particular importance in the `ChildSpec` tuple is the child `Id`. Even after termination, the `ChildSpec` will be stored by the supervisor and referenced through its `Id`, allowing processes to stop and restart the child. Only upon deletion will the child specification be permanently removed.

If you've been skimming through the manual page for the supervisor behavior, you probably realize that it does not export a `stop` function. As supervisors are never meant to be stopped by anyone other than their parent supervisors, this function was not implemented.

You can easily add your own `stop` function by including the following code in your supervisor callback module. This will however work only if `stop` is called by the parent:

```
stop() -> exit(whereis(?MODULE), shutdown).
```

If your supervisor is not registered, use its pid.

Applications

The *application* behavior is used to package Erlang modules into reusable components. An Erlang system will consist of a set of loosely coupled applications. Some are developed by the programmer or the open source community, and others will be part of the OTP distribution. The Erlang runtime system and its tools will treat all applications equally, regardless of whether they are part of the Erlang distribution.

There are two kinds of applications. The most common form of applications called *normal applications*, will start the supervision tree and all of the relevant static workers. *Library applications* such as the Standard Library, which come as part of the Erlang distribution, contain library modules but do not start the supervision tree. This is not to say that the code may not contain processes or supervision trees. It just means they are started as part of a supervision tree belonging to another application.

In this section, we will cover all the functionality needed to encapsulate the mobile subscriber system into an OTP application, starting its top-level supervisor. When done, this application will behave like any other normal application. And don't forget, when we talk about applications in this chapter, we mean OTP applications.

Applications are loaded, started, and stopped as one unit. A resource file associated with every application not only describes it, but also specifies its modules, registered processes, and other configuration data. Applications have to follow a particular directory structure which dictates where beam, module, resource, and include files have to be placed. This structure is required for many of the existing tools, built around behaviors, to function correctly. To find out which applications are running in your Erlang runtime system, you use `application:which_applications()`:

```
1> application:which_applications().
[{stdlib,"ERTS  CXC 138 10","1.15.2"},
 {kernel,"ERTS  CXC 138 10","2.12.2"}]
```

The Standard Library and the Kernel are part of the basic Erlang applications and together form the minimal OTP subset when starting the runtime system. The first item in the application tuple is the application name. The second is a description string, and the third is the application version number. If you are wondering what the description string in the preceding example means, you are not alone. It is the internal Ericsson product numbering scheme.

We will show you where to configure the description of your applications later in this chapter.

Directory Structure

In your Erlang shell, type `code:get_path()`. You did this when we were explaining how to manipulate the code search path in the code server. What you probably did not realize at the time was that each code path was pointing to a specially structured directory of an OTP application.

Let's pick the Inets application and inspect its contents in more detail. In Windows, the path for this particular installation of Erlang would be as follows:

```
C:/Program Files/erl5.6.2/lib/inets-5.0.5/
```

In other operating systems, just `cd` to the *lib* directory from the Erlang root directory, typically something like */usr/local/lib/erlang/lib*, and look for the latest Inets release. Among all the subdirectories in an application, the following ones comprise an OTP release of the application in question:

src

Contains the source code of all the Erlang modules in the application.

ebin

Contains all of the compiled beam files and the application resource file; in this example, it's *inets.app*.

include

Contains all the Erlang header files (`hrl`) intended for use outside the application. By using the following directive,

```
-include_lib("Application/include/Name.hrl")
```

where `Application` is the application directory name *without* the version number (in the example it would be `inets`) and `Name.hrl` is the name of the include file, the compiler will automatically pick up the version of the application pointed to by the code search path.

priv

Is an optional directory that contains necessary scripts, graphics, configuration files, or other non-Erlang-related resources. You can access it without knowing the application version by using the `code:priv_dir(Application)` call.

You will notice that Inets (and other) applications may have a few more directories, including *docs* and *examples*. These have no effect on the system during runtime, and are there just for convenience. In some applications, you might not find the *priv* directory. If you do not use it, omitting it is not a problem, even if omitting it might not be considered a good practice by some. In live systems, the only mandatory directory is *ebin*. This is because you probably don't want to include your source code when shipping your system to clients!

It is common to use scripts to create these directory structures, and to use make files which, having compiled your code, move the beam files to the *ebin* directory. How you set this up depends on the operating systems, build systems, repositories, and many other non-Erlang-related dependencies in your application. Although it might be feasible to set this up manually for small projects, you will probably want to use templates and automate the task for larger projects.

The Application Resource File

The application resource file, also known as the *app file*, contains information on your application resources and dependencies. Move into the *ebin* subdirectory of the Inets application and look for the *inets.app* file. This is the resource file of the Inets application. On closer inspection, you will notice that all other applications also have an *inets.app* file. The application resource file consists of a tuple where the first element is the application tag, the second is the application name, and the third is a list of features.

Let's go through the features individually. Note that for space considerations, we've omitted some of the modules in the example.

```
{application,inets,
 [{description,"INETS  CXC 138 49"},
  {vsn,"5.0.5"},
  {modules,[inets,inets_sup,inets_app,inets_service,
            %% FTP
            ftp, ftp_progress,ftp_response,ftp_sup,
            %% HTTP client:
            http,httpc_handler,httpc_handler_sup,httpc_manager,
            %% TFTP
            tftp,tftp_binary,tftp_engine,tftp_file,tftp_lib,tftp_sup
          ]},
  {registered,[inets_sup, httpc_manager]},
  {applications,[kernel,stdlib]},
  {mod,{inets_app,[]}}]}.
```

In the preceding code, the `description` is a string that is displayed as a result of calling the `application:which_application/0` function. The `vsn` attribute is a string denoting the version of the application. This should be the same as the suffix of the application directory. In larger build systems, the application version is usually updated automatically through proprietary scripts executed when committing your code.

The `modules` tag lists all the modules that belong to this application. The purpose of listing them is twofold. The first is to ensure that all of them are present when building the system and that there are no name clashes with any other applications. The second is to be able to load them either at startup or when loading the application. For every module, there should be a corresponding beam file. To ensure that there are no registered name clashes with other applications, we list all of the `registered` processes in this field. Clashes in module and registered process names are detected by the release-handling tools used when creating your boot file. We will look at boot files in the next section. Just including them in the application resource files will have no effect unless these tools are used.

Most applications will have to be started after other applications on which they depend. Your application will not start if the applications in the `applications` list included in your *app* file are not already started. *kernel* and *stdlib* are the basic standard applications on which every other application depends. After that, the particular dependencies will be based on the nature of the application.

Finally, the `mod` parameter is a tuple containing the callback module and the arguments passed to the `start/2` callback function.

Not necessary to the Inets application, but certainly important to applications in general, are *environment variables*. The `env` tag indicates a list of key-value tuples that can be accessed from within the application using calls to the following functions:

```
application:get_env(Tag)
application:get_all_env().
```

To access environment variables belonging to other applications, just add the application `Name` to either function call, as in the following:

```
application:get_env(Name,Tag)
application:get_all_env(Name).
```

The application resource file *usr.app* of our mobile subscriber service database would contain four modules, two registered processes, and dependencies on the *stlib* and *kernel* applications. Let's also add the filename for the Dets table among the environment variables:

```
{application, usr,
 [{description, "Mobile Services Database"},
  {vsn, "1.0"},
  {modules, [usr, usr_db, usr_sup, usr_app]},
  {registered, [usr, usr_sup]},
  {applications, [kernel, stdlib]},
  {env, [{dets_name, "usrDb"}]},
  {mod, {usr_app,[]}}]}.
```

Starting and Stopping Applications

You start and stop applications using the following commands:

```
application:start(ApplicationName).
application:stop(ApplicationName).
```

In the preceding code, `ApplicationName` is an atom denoting the name of your application.

The *application controller* loads the environment variables belonging to the application, as well as starts the top-level supervisor through a set of callback functions. When calling `start/1`, the `start(StartType, Arguments)` function in the application callback module is invoked. `StartType` is usually the atom `normal`, but if you are dealing with distributed applications,[2] you might come across the `start` types `takeover` and `failover`. `Arguments` is a value of any valid Erlang data type, which together with the callback module is defined in the application resource file.

`Start` has to return the tuple `{ok, Pid}` or `{ok, Pid, Data}`. `Pid` denotes the process identifier of the top-level supervisor. `Data` is a valid Erlang data type used to store data that is needed when terminating the application.

If you stop your application, the top-level supervisor is sent a shutdown message. This results in the termination of all of its children in reverse startup order, propagating the exit path through the supervision tree. Once the supervision tree has terminated, the callback function `stop(Data)` is called in the application callback module. `Data` was originally returned in the `{ok, Pid, Data}` construct of the `start/2` callback function. If your `start/2` function did not return any data, just ignore the argument. Should you want a callback function to be called before terminating the supervision tree, export the function `prep_stop(Data)` in your callback module.

So, armed with all of the preceding information, how would you package your `usr` server database into an application, what would the directory structure look like, and what are the contents of the app file?

[2] We do not cover distributed applications in this chapter. For more information on them, you will need to consult the OTP documentation.

Let's start with the application callback file. We export the `start/2` and `stop/1` functions:

```
-module(usr_app).
-behaviour(application).
-export([start/2, stop/1]).

start(_Type, StartArgs) ->
    usr_sup:start_link().
stop(_State) ->
    ok.
```

As you can see, the application callback module is relatively simple. Although we have not done it in our example, it is not uncommon to join the supervisor and application behavior modules into one. You would have the two `-behaviour` directives next to each other, and if there is no conflict with the callback functions, the compiler will not issue any warnings.

This leaves one minor change to be made in the `usr.erl` module, where we read the environment variable in the `start_link/0` call:

```
start_link() ->
    {ok, FileName} = application:get_env(dets_name),
    start_link(FileName).
```

With all of this in place, all that remains is our application directory structure, placing the relevant files in there:

```
usr-1.0/src/usr.erl
            usr_db.erl
            usr_sup.erl
            usr_app.erl
       /ebin/usr.beam
             usr_db.beam
             usr_sup.beam
             usr_app.beam
             usr.app
       /priv/
       /include/usr.hrl
```

Let's compile all the modules and take them for a test run. Move the beam files to the *ebin* directory, and make sure they are accessible by telling the system about the path to them. You can do that either with the `erl -pa Dir` directive when starting Erlang, or directly in the shell using `code:add_path(Dir)`.

In the following interaction, we start the application and run a few operations on the customer settings before stopping it. In doing so, we check that the `supervisor` and `gen_server` processes no longer exist.

```
1> code:add_path("usr-1.0/ebin").
true
2> application:start(usr).
ok
3> application:start(usr).
{error,{already_started,usr}}
4> usr:lookup_id(10).
{error,instance}
5> application:get_env(usr, dets_name).
```

```
{ok,"usrDb"}
6> application:stop(usr).

=INFO REPORT==== 27-Jan-2009::22:14:33 ===
    application: usr
    exited: stopped
    type: temporary
ok
6> whereis(usr_sup).
undefined
```

Note how we retrieved the `dets_name` environment variable from the environment. In our `usr` example, we are calling the function from within the application, and as a result, we do not need to specify the application name. Look through the manual page of the application module, and experiment with the various options for retrieving application environment variables.

The Application Monitor

The application monitor is a tool that provides an overview of all running applications. Upon launching it with the `appmon:start()` call, you are presented with a list of all the applications running on all distributed nodes. The various menus allow you to manipulate the node and presentation, and the bar on the left shows the load on the node under scrutiny (see Figure 12-3).

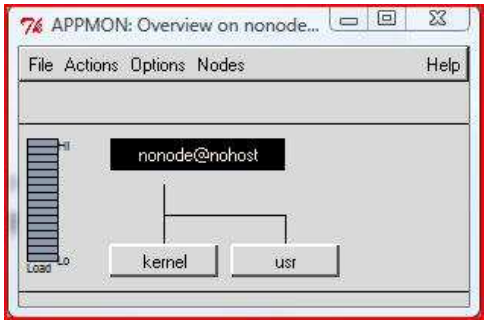

Figure 12-3. The application monitor window

In Figure 12-3, note how the *stdlib* application is not shown. Only applications with a supervision tree appear. Double-clicking the application opens a new window with a view of its supervision tree (see Figure 12-4). The menus and buttons allow you to manipulate the various processes. The top processes linking to `usr_sup` are part of the application controller. They are the ones that start, monitor, and stop the top-level supervisor.

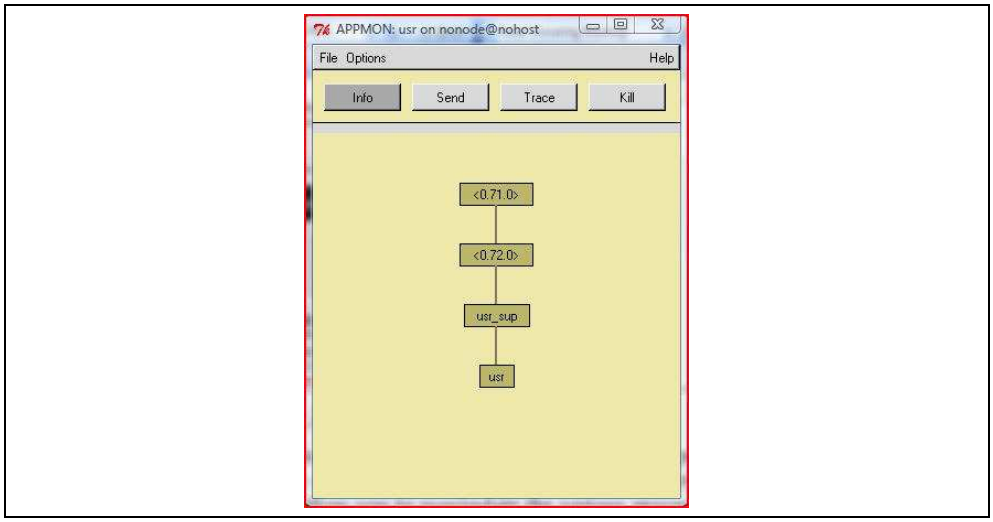

Figure 12-4. The supervision tree viewed in the application monitor

Release Handling

From our behaviors, we've created a supervision tree. The supervision tree is packaged in an application that can be loaded, started, and stopped as one entity. Erlang systems consist of a set of loosely coupled applications specified in a release file. This includes the basic Erlang installation you have been running. From your Erlang root directory, enter the *releases* directory, followed by one of the release subdirectories. In our example, it is *R12B* (see Figure 12-5).

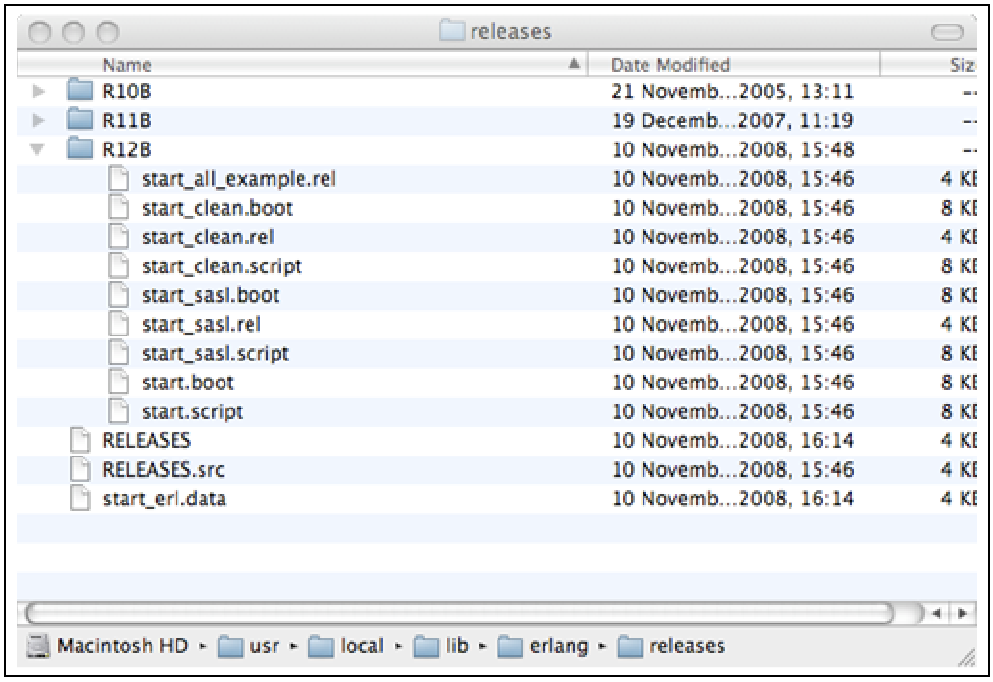

Figure 12-5. The releases directory

In it, you will find a list of release files, indicated by the *.rel* suffix, as shown in Figure 12-5. Pick *start_clean.rel* and inspect it:

```
{release, {"OTP  APN 181 01","R12B"}, {erts, "5.6.2"},
 [{kernel,"2.12.2"},
  {stdlib,"1.15.2"}]}.
```

It consists of a tuple where the first element is the `release` tag and the second element is a tuple with a release name and release version number. The third element is a tuple with the version of the Erlang runtime system. The last element in the tuple is a list of applications and their version numbers, defined in the order in which they should be started.

Each application in this list points to an application resource file. When you call the function `systools:make_rel(Name, Options)`, these app files are retrieved and inspected. Module and registered process name conflicts are checked, and if everything matches, *Name.boot* and *Name.script* files are produced.

Name.boot is a binary file containing instructions on loading the application modules and starting the top-level supervisors. The *Name.script* file is a text version of its binary counterpart. The `Options` argument is a list in which the most important data includes `{path, [Dir]}`, which describes any paths to the application *ebin* directories not known by the code server. The paths commonly point to the *ebin* directories of your applications. The `local` directive is another option to `make_rel/2`, stating that the boot file should not assume that all the applications will be found under *lib* in the Erlang root directory. The latter is useful if you want to separate the Erlang installation and your applications.

In a deployed system, you will release only the applications that are relevant to your system. These applications include both your applications and the subset of the ones you need from the OTP release. They should all be stored in the *lib* subdirectory of the Erlang root, but you can easily override this recommendation by adding paths in the code search path used by the code server. We are in fact overriding this in our example by using the `local` directive in the options list.

When the boot file has been created, you can start your system using the following command:

```
erl -boot Name
```

This ensures that all of the modules specified in the boot file are loaded and that the applications and their respective supervision trees are started correctly. If anything fails at startup, the Erlang node will not start.

The release file of our mobile subscriber database, *usr.rel*, would include *kernel* and *stdlib*, the two mandatory applications of any OTP release, together with version 1.0 of our `usr` application:

```
{release, {"Mobile User Database","R1"}, {erts, "5.6.2"},
 [{kernel,"2.12.2"},
  {stdlib,"1.15.2"},
  {usr,"1.0"}]}.
```

Because in the running example we are using the shell where we previously added the path to `usr-1.0/ebin`, we create a boot file which runs with the existing code path. Had we *not* set the path in the shell, we would have had to add the option `{dir, ["usr-1.0/ebin"]}` to the release file.

```
7> systools:make_script("usr", [local]).
ok
8> ls().
usr-1.0
usr.boot
usr.rel
usr.script
usrDb
```

We can now start our system using `erl -boot usr`.

> Have a look at the *usr.script* file that was generated when creating the boot file. We will not explain it in this book, as most of its commands should be fairly straightforward. You can edit the files and generate a new boot file using the `systools:script2bootfile/1` call.
>
> Spare a thought for the early pioneers of OTP. In its first release back in 1996, script files had to be generated manually, as the `make_script/2` function had not been implemented!

Other Behaviors and Further Reading

What we described in this chapter should cover a majority of the cases you will come across when using OTP behaviors. However, you can go in more detail when working with generic servers, supervisors, and applications. Behaviors we have not covered but

which we briefly introduced in this chapter include finite state machines (finite LoopData machines), event handlers, and special processes. All of these behavior library modules have manual pages that you can reference. In addition, the Erlang documentation has a section on OTP design principles which provides more details and examples of these behaviors.

Finite state machines are a crucial component of telecom systems. In Chapter 5, we introduced the idea of modeling a phone as a finite state machine. If the phone is not being used, it is in state idle. If an incoming call arrives, it goes to state ringing. This does not have to be a phone; it could instead be an ATM cross-connect or the handling of data in a protocol stack. The gen_fsm module provides you with a finite state machine behavior which you can use to solve these problems. States are defined as callback functions that return a tuple containing the next State and the updated loop data. You can send events to these states both synchronously and asynchronously. The finite state machine callback module should also export the standard callback functions such as init, terminate, and handle_info. As gen_fsm is a standard OTP behavior, it can be linked to the supervision tree.

Event handlers and managers are another behavior implemented in the gen_event library module. The idea is to create a centralized point that receives events of a specific kind. Events can be sent both synchronously and asynchronously with a predefined set of actions being applied when they are received. Possible responses to events include logging them to file, sending off an alarm in the form of an SMS, or collecting statistics. Each of these actions is defined in a separate callback module with its own LoopData, preserved in between calls. Handlers can be added, removed, or updated for every specific event manager. So, in practice, for every event manager, there could be many callback modules, and different instances of these callback modules could exist in different managers.

Sometimes you might want to add to the supervision tree processes that are not generic OTP behaviors. This might be for efficiency reasons, where you have implemented the process using plain Erlang. You might want to attach to a supervision tree legacy code that was written before OTP was available, or you might have abstracted a design pattern and implemented your own behavior.

Writing your own behaviors is straightforward. The main differences are in how you spawn your processes, and the system calls you need to handle. You should create processes using the proc_lib library, which exports both spawn and start functions. Using the proc_lib function stores the start data of the process, provides the means to start the process synchronously, and generates error reports upon abnormal termination. To be OTP-compliant, processes need to handle system messages and events, yielding the control of the loop to the sys library module. They also need to be linked to their parent, and if they are trapping exits, they need to terminate when the parent terminates. You can find more information on writing your own OTP behaviors in the sys and proc_lib library modules.

Exercises

Exercise 12-1: Database server revisited

Rewrite Exercise 5-1 in Chapter 5 using the `gen_server` behavior module. Use the `lists` backend database module, saving your list in the loop data. You should register the server and access its services through a functional interface. Exported functions in the `my_db_gen.erl` module should include the following:

```
my_db_gen:start() ⇒ ok.
my_db_gen:stop() ⇒ ok.
my_db_gen:write(Key,Element) ⇒ ok.
my_db_gen:delete(Key) ⇒ ok.
my_db_gen:read(Key) ⇒ {ok,Element}|{error,instance}.
my_db:match(Element) ⇒ [Key1, ..., KeyN].
```

Hint: if you are using Emacs or Eclipse, use the `gen_server` skeleton template:

```
1> my_db:start().
ok
2> my_db:write(foo, bar).
ok
3> my_db:read(baz).
{error, instance}
4> my_db:read(foo).
{ok, bar}
5> my_db:match(bar).
[foo]
```

Exercise 12-2: Supervising the database server

Implement a supervisor that starts and monitors the `gen_server` in Exercise 12-1. Your supervisor should be able to handle five crashes per hour. Your child should be permanent and be given at least 30 seconds to terminate, as it might take some time to close a large Dets file.

Exercise 12-3: The database server as an application

Encapsulate your supervision tree from Exercise 12-2 in an application, setting up the correct directory structure, complete with application resource file.