

CS11 – Erlang

Winter 2008–2009

Lecture 7

Storage in Erlang

- ▶ Erlang provides several mechanisms for storing and retrieving data
- ▶ Today:
 - Simple file-based storage
 - ETS: Erlang Term Storage
 - DETS: Disk ETS
- ▶ All of these mechanisms support storage of Erlang terms
 - Literals, lists, tuples, etc.
 - Can also store arbitrary chunks of data

Erlang File IO

- ▶ File operations are provided in four modules
- ▶ **file** module:
 - Wide variety of most common tasks
 - Open files, close files, delete files
 - Retrieve file info, directory info, symbolic link info
 - Change directory, list directory contents, etc.
- ▶ **filename** module:
 - Functions for manipulating filenames in a reasonably platform-independent way (similar to Ant capabilities)
 - Combine paths, normalize paths, get absolute paths
 - Get directory/filename/extension from a path
 - Break apart a path into components

Erlang File IO (2)

- ▶ **filelib** module:
 - Utility functions for files and directories
 - Test if a path is a file or a directory, etc.
 - Perform wildcard matching on files
 - Recursively traverse directory hierarchies
- ▶ **io** module:
 - Functions for parsing input data in various ways
 - Functions for formatting output
 - Have been using this frequently, but specifically designed for file IO

Reading Erlang Terms

- ▶ Simplest way to read a file:
 - `file:consult(Filename)`
- ▶ Reads a file containing Erlang terms, each of which is terminated with “.”
 - File can also contain Erlang comments
- ▶ Input is a string filename and path
- ▶ Result is `{ok, Terms=[term()]}`
 - Each term in config file is an element in the list
 - (Or, various kinds of error information...)

file:consult() Example

▶ Example: a list of users users.erl

```
% List of users, including usernames, description,  
% and registration date.  
{user, donnie, "Donnie Pinkston",  
  {{2007,5,15}, {9,35,21}}}. % May 15, 2007 9:35 AM  
{user, mvanier, "Mike Vanier",  
  {{2007,7,20}, {14,26,53}}}. % Jul 20, 2007 2:26 PM
```

▶ To load file:

- {ok, Contents} = file:consult("users.erl").
- Contents is a list with two elements
 - First element is {user, donnie, ...}
 - Second element is {user, mvanier, ...}

`file:consult()` Limitations

- ▶ Contents of input file may only include:
 - Literal values, Tuples, Lists
- ▶ Can't specify Funs, PIDs, Refs, etc.
- ▶ Also can't specify records
 - At least, not with `#record(attr1=val1, ...)` syntax
 - Instead, specify the tuple version in the input file:
`{record, val1, ...}`
 - Can still use record syntax in your processing code
- ▶ If you use record structures in the input file, make sure to document them!
 - Don't have benefit of field names in the file...

Opening Files

- ▶ Most other file IO operations require you to open the file before working with it

`file:open(Filename, Modes)`

- `Filename` is simply a path
- `Modes` is a list of atoms specifying file mode
 - Simple modes: `read`, `write`, `append`
 - Optimize file IO operations: `read_ahead`, `delayed_write`
 - Tell Erlang to read extra input from the file, or batch up write operations
 - Read or write gzip-compressed files: `compressed`
 - Raw data: `raw`, `binary`
- ▶ Example:

```
{ok, F} = file:open("users.erl", [read, write]).
```

Opening Files (2)

- ▶ Example:

 - `{ok, F} = file:open("users.erl", [read, write]).`

- ▶ **F** is an IoDevice

 - As long as `raw` isn't specified, is actually a *process* that interacts with the file
 - IO process can interact with file in background
 - IoDevice process is linked to opening process
 - When opening process terminates, open files are automatically closed

- ▶ If `raw` is specified, **F** is not a process

 - Much more efficient IO, but also very limited features
 - Can only use low-level read/write operations
 - More about `raw` in a bit...

- ▶ Closing a file is simple: `file:close(IoDevice)`

Reading Erlang Terms

- ▶ `io:read()` reads individual Erlang terms from an open file
 - `io:read(IoDevice, Prompt)`
 - `io:read(Prompt)` for reading from standard input
 - `Prompt` is only used when reading from standard input
 - When reading from a file, specify a dummy value

- ▶ Example:

```
1> {ok, Users} = file:open("users.erl", [read]).
{ok,<0.33.0>}
2> io:read(Users, '').
{ok, {user,donnie,"Donnie Pinkston", ...}}
3> io:read(Users, '').
{ok, {user,mvanier,"Mike Vanier", ...}}
4> io:read(Users, '').
eof
```

Writing Erlang Terms

- ▶ `io:write()` can write individual terms

- `io:write(IoDevice,Term)`
- `io:write(Term)` for writing to standard output
- Doesn't provide very good formatting

- ▶ Example:

```
io:write({user, donnie, "Donnie Pinkston", ...}).  
{user,donnie,[68,111,110,110,105,...], ...}
```

- No period at end of term; strings written as numbers!

- ▶ Much better solution: `io:format()`

```
io:format("~p.~n",  
          [{user, donnie, "Donnie Pinkston", ...}] ).  
{user,donnie,"Donnie Pinkston",...}.
```

- Wraps output at reasonable places to avoid long lines
- Also `io:fwrite()`, which is identical to `io:format()`

Reading and Writing Lines of Text

- ▶ Read entire lines of text using `io:get_line`
 - `io:get_line(IoDevice, Prompt)`
 - `io:get_line(Prompt)` for reading from std input
 - Reads up to newline character
 - Returns the string, *including* newline character, or `eof` if end of file
- ▶ Again, use `io:format()` or `io:fwrite()` to write lines
 - `io:format(File, "~s~n", [Line]).`

Reading an Entire File

- ▶ Can use `file:read_file(Filename)` to read an entire file in one shot
 - Most efficient way of retrieving a file's contents
 - Of course, entire file must fit into memory...
- ▶ Returns `{ok, Binary}`
 - `Binary` is a raw, untyped chunk of data
- ▶ Example:

```
1> file:read_file("users.erl").  
{ok,<<"% List of users, including username  
s, description,\n% and registration date.\n{user, donnie, \"Donnie Pinkston\"...>>}
```

Erlang Binary Datatype

- ▶ Erlang has binary datatype for untyped blocks of data
- ▶ Binary data specified, matched with bit expressions
 - `<<E1, E2, ...>>`
- ▶ **E_i** is:
 - **Value** – a literal integer, float, or another binary
 - **Value:Size** – **Size** is an integer in “units”
 - For integers: default unit = 1 bit, default size = 8
 - For floats: default unit = 1 bit, default size = 64
 - For binary, default unit = 8 bits, default size is entire binary
- ▶ **Examples:**
 - `<<68,111,110,110,105,101>>`
 - Also `<<"Donnie">>` or `<<$D,$o,$n,$n,$i,$e>>`
 - `<<1,17,42:16>>`
 - Evaluates to `<<1,17,0,42>>`

Erlang Binary Datatype (2)

- ▶ E_i can also be:
 - `Value/TypeSpecifierList`
 - `Value:Size/TypeSpecifierList`
- ▶ Type specifier lists are concatenation of atoms separated by hyphens, specifying:
 - Type (`integer`, `float`, `binary`) - default `integer`
 - Signedness (`signed`, `unsigned`) - default `signed`
 - Endianness (`big`, `little`, `native`) - default `big`
 - Unit - value in range of 1..256
 - Defaults to 1 for integer/float, 8 for binary
- ▶ Simple examples:
 - `<<3.14159/float>>`
 - Evaluates to `<<64,9,33,249,240,27,134,110>>`
 - `<<3.14159/float-little>>`
 - Evaluates to `<<110,134,27,240,249,33,9,64>>`

Matching with Binaries

- ▶ Can use unbound variables in a binary specification
 - Use to break down an existing binary
- ▶ Cool example from Erlang docs:
 - Break down an IPv4 datagram packet

```
-define(IP_VERSION, 4).  
-define(IP_MIN_HDR_LEN, 5).
```

```
DgramSize = size(Dgram),  
case Dgram of  
  <<?IP_VERSION:4, HLen:4, Srvctype:8, TotLen:16,  
    ID:16, Flgs:3, FragOff:13,  
    TTL:8, Proto:8, HdrChkSum:16,  
    SrcIP:32,  
    DestIP:32, RestDgram/binary>> when HLen>=5, 4*HLen=<DgramSize ->  
    OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),  
    <<Opts:OptsLen/binary,Data/binary>> = RestDgram,  
    ...  
end.
```

- Match literal values in binary data
- Extract nibbles, other non-byte-size values
- Remainder of packet goes into **RestDgram**

BIFs for Binaries

- ▶ **is_binary(Term)**
 - Returns true/false if term is a binary
- ▶ **size(Bin)**
 - Returns the number of bytes in the binary
- ▶ **split_binary(Bin, Pos)**
 - Splits a binary into two binaries
- ▶ **binary_to_list(Bin), list_to_binary(List)**
 - Convert between a binary and a sequence of bytes
- ▶ **binary_to_list(Bin, Start, Stop)**
 - Convert a subrange of a binary to a character sequence
- ▶ **binary_to_term(Bin), term_to_binary(Term)**
 - Convert between a binary and an Erlang term

Raw File IO

- ▶ Can also specify **raw** mode to **file:open()**
- ▶ Result is *not* a process!
 - Actually a file-descriptor for direct local-file access
- ▶ Cannot use **io** functions to access file!
 - Can use **file:read()** and **file:write()** to access file
 - (These functions also work with normal files)
- ▶ **file:read(IODevice, Num)**
 - Reads up to **Num** bytes from file
 - If at least one byte read, result is a binary
 - (If file not opened in **raw** mode, result is a string.)
 - If EOF before any bytes are read, returns **eof**
- ▶ **file:write(IODevice, Bytes)**
 - Writes the specified data to the file

Random-Access File IO

- ▶ Erlang files also support random access
 - `file:position(IoDevice, Location)`
 - `Location` can be an absolute offset, or a tuple
 - `{bof, Offset}` - offset from beginning of file
 - `{cur, Offset}` - offset from current position
 - `{eof, Offset}` - offset from end of file
 - Other variations too
 - Also `file:pread()` and `file:pwrite()`
 - Combines position and read/write operations into one function
- ▶ These work on both normal files and raw files

The `iodata()` and `iolist()` Types

- ▶ Most write-file functions can handle more than just binary or list data
 - Specified to take `iodata()` or `iolist()` types
- ▶ Example: `file:write(IODevice, Bytes)`
 - `Bytes` is of type `iodata()`
- ▶ Definitions:
 - `iodata()` = `binary()` | `iolist()`
 - `iolist()` = [`char()` | `binary()` | `iolist()`]
- ▶ `iodata/iolist` can be lists of binaries, lists of lists of data, any level of nesting
 - Structures get flattened before writing to the file
 - e.g. `list_to_binary(IoList)` flattens input `iolist` data

Using `iodata()` and `iolist()` Types

- ▶ Nested `iolist()` values are sometimes useful
- ▶ Example: HTML/XML output

```
tag_data(Tag, Data) when is_atom(Tag) ->  
  TagStr = atom_to_list(Tag),  
  ["<" ++ TagStr ++ ">", Data, "</" ++ TagStr ++ ">"].
```

- Data and markup are separate in internal representation
- ▶ Use to construct HTML:

```
Page = tags:tag_data(html,  
  tags:tag_data(body,  
    tags:tag_data(h1, "Welcome to CS11!")))).
```

- Result is:

```
["<html>",  
  ["<body>",  
    ["<h1>", "Welcome to CS11!", "</h1>"],  
    "</body>"],  
  "</html>"]
```

HTML Example (2)

- ▶ Our HTML page:

```
Page = tags:tag_data(html,  
    tags:tag_data(body,  
        tags:tag_data(h1, "Welcome to CS11!"))).  
["<html>",  
    ["<body>",  
        ["<h1>", "Welcome to CS11!", "</h1>"],  
        "</body>"],  
    "</html>"]
```

- ▶ Easy to output to file, console, etc.

```
io:format("~s~n", [Page]).
```

```
<html><body><h1>Welcome to CS11!</h1></body></html>
```

- Output is automatically flattened to a single list

Erlang Term Storage

- ▶ ETS and DETS are two modules in Erlang for storing and accessing large sets of terms
 - Very large key-value lookup tables
- ▶ ETS = Erlang Term Storage
 - Data is stored in memory only
 - When program shuts down, the data goes away
 - Very fast, but all your data is in memory
- ▶ DETS = Disk ETS
 - Data is stored on disk, so it persists through shutdowns
 - Significantly slower than ETS due to disk-access
 - Memory footprint much lower than ETS since only some of the data is kept in memory

ETS Tables

- ▶ ETS tables store tuples
- ▶ One value in the tuple is the key
 - Used to identify the entire tuple
 - By default, is the first value in the tuple
- ▶ Internal storage/lookup mechanism depends on type of ETS table
 - **set** – each key may only appear once
 - Uses a hashtable for storage/lookup
 - Constant-time lookup/insertion
 - **ordered_set** – same as set
 - Tuples kept in sorted order based on key
 - Logarithmic-time lookup/insertion
 - Can traverse table contents in sorted order very easily

ETS Tables (2)

- ▶ Two more types of ETS table:
 - **bag**
 - A key can appear multiple times, but each tuple is still unique
 - e.g. can write $\{a,1\}$ to the table, then $\{a,3\}$
 - Table will contain $[\{a,1\}, \{a,3\}]$
 - Writing $\{a,1\}$ again doesn't add another row
 - Table still contains only $[\{a,1\}, \{a,3\}]$
 - **duplicate_bag**
 - Like bag, a key can appear multiple times
 - A tuple can also appear multiple times in the table
 - Writing $\{a,1\}$ to table containing $[\{a,1\}, \{a,3\}]$ produces:
 - $[\{a,1\}, \{a,3\}, \{a,1\}]$

ETS Table Efficiency Notes

- ▶ Need to pick table type most suitable for task
 - Sets, bags, duplicate bags all use hash tables
 - Ordered sets use balanced binary trees
- ▶ Ordered sets are slower than sets
 - Only use if you need in-order traversal of data
- ▶ Bags are slower than duplicate bags
 - Duplicate tuples are excluded at insert time
 - ETS has to check all existing tuples with specified key value, when writing a new tuple to the table
 - Duplicate bags don't require this overhead

ETS Table Efficiency Notes (2)

- ▶ All Erlang terms are copied to/from ETS tables...
 - Avoid looking up data from ETS when possible
- ▶ One exception: large binaries are stored outside of the normal Erlang heap
 - Storing large binaries into ETS tables is very fast
 - Retrieving large binaries from ETS tables is very fast
 - Passing large binaries between processes also very fast
- ▶ Consider using binaries instead of large strings, or large complex data structures
 - Of course, if this impacts readability/correctness, then use the slower, easier to understand version!

Creating an ETS Table

- ▶ Use `ets:new(Name, [Opts])`
 - Specify table type in options-list
 - Returns a TableId for accessing the new table
- ▶ Can specify various options:
 - `private` - table is only accessible by the process that creates it
 - `protected` - only the owner can write to the table
 - Other processes can read from the table
 - `public` - any process can write to the table
 - `{keypos, K}` - use value at position K (1-based)
- ▶ Specifying an empty options-list is same as:
 - `[set, protected, {keypos, 1}]`

Inserting Tuples

- ▶ Use `ets:insert()` to add tuples to a table
 - `ets:insert(TableId, tuple())`
 - `ets:insert(TableId, [tuple()])`
 - Always returns true
 - `TableId` can be name (atom) or ID returned by `new()`
- ▶ Tuple may or may not be added, depending on table type
 - Tuples will always be added to duplicate bag
 - For sets, ordered sets, bags, tuple may replace an existing tuple, or may already be in table
- ▶ To only insert tuples with new key values:
 - `ets:insert_new(TableId, tuple())`
 - `ets:insert_new(TableId, [tuple()])`
 - Only inserts tuple(s) if no key-value already appears in table

ETS Example: User Database

```
% Load the user list from the data file.
{ok,Users} = file:consult("users.erl").

% Key is in 2nd position, since we have user
% records: {user,donnie,"Donnie Pinkston",...}
UserTab = ets:new(user_table,
    [ordered_set,protected,{keypos,2}]).

% Add all user records to the ETS table.
ets:insert(user_table, Users).

% Can also do ets:insert(UserTab, Users)
```

Looking Up Tuples

- ▶ Use `ets:lookup(Tab, Key)` to retrieve tuples
 - Result is a list of tuples
- ▶ If table is a set or ordered set, result will contain 0 or 1 tuples
- ▶ If table is a bag or duplicate bag, result could be 0 or more tuples
- ▶ Example:
 - `ets:lookup(UserTab, donnie)`.
 - Returns `[{user, donnie, "Donnie Pinkston", ...}]`
- ▶ `ets:member(Tab, Key)` for membership tests
 - Looks up key, but doesn't retrieve any tuples

Traversing an ETS Table

- ▶ `ets:first(Tab)` returns first key in table
- ▶ `ets:next(Tab, Key)` returns next key
 - `'$end_of_table'` atom returned if end of ETS table is reached
- ▶ `ets:prev(Tab, Key)` returns previous key
 - Also returns `'$end_of_table'` when table start reached
- ▶ `ets:last(Tab)` returns last key in table
- ▶ Keys will/will not be in increasing order, based on type of table
- ▶ Can use the keys with `ets:lookup()` to retrieve the actual tuples

Deleting Tuples and Tables

- ▶ Several functions to delete tuples
- ▶ **ets:delete(TableId, Key)**
 - Deletes all tuples with the specified key
- ▶ **ets:delete_object(TableId, Tuple)**
 - Deletes a specific object in the ETS table
- ▶ **ets:delete_all_objects(TableId)**
 - Deletes all objects in the ETS table
 - This is an atomic operation
- ▶ **ets:delete(TableId)**
 - Deletes the entire ETS table
 - Can't be used after it's deleted

Searching for Tuples

- ▶ Two main functions to find tuples
- ▶ `ets:match(TableId, Pattern)`
 - Pattern is a match specification
 - An Erlang term comprised entirely of literals
 - Specifies which records to retrieve, and what parts of records
- ▶ Some simple examples:
 - `ets:match(UserTab, '$1')`.
 - Returns list of all user records
 - `ets:match(Users, {user, '_', '$1', '_'})`.
 - Returns the display-name of all users in the table
 - `ets:match(Users, {user, '_', "Donnie Pinkston", '$1'})`.
 - Returns the registration date of the user with a display-name of "Donnie Pinkston"

Searching for Tuples (2)

- ▶ Variants of match:
 - `ets:match_delete(Tab, Pattern)`
 - Deletes all tuples that match the pattern
 - `ets:match_object(Tab, Pattern)`
 - Returns the list of all objects that match the pattern
 - `ets:match_object(Tab, Pattern, Limit)`
 - Returns up to `Limit` objects in result
- ▶ Also `ets:select(Tab, MatchSpec)`
 - Generalized version of `ets:match()` that supports lists of patterns, very sophisticated tests, etc.
 - Also variants: `select_delete()`, `select_count()`

DETS: Disk Erlang Term Storage

- ▶ Very similar to ETS tables
- ▶ DETS files can grow up to 2GB in size
- ▶ DETS supports `set`, `bag`, and `duplicate_bag`
 - No `ordered_set` support in DETS!
- ▶ DETS tables must be properly closed
 - DETS module will detect when table wasn't closed properly
 - Main issues:
 - May lose last records written to table
 - DETS will perform a check on every record to restore table to a valid state. This could be very slow...
- ▶ Mnesia database implemented on top of DETS

Opening/Creating DETS Tables

- ▶ Use `dets:open_file()` to open an existing table, or create a new table
- ▶ `dets:open_file(Name, Args)`
 - Open a DETS table file. If file doesn't exist, create.
 - `Args` are much more sophisticated than ETS tables, for obvious reasons
 - File permissions, estimated size/usage details, force repair, caching behaviors, etc.
- ▶ `dets:open_file(Filename)`
 - Open an existing DETS file. Don't have to specify all that other junk...
 - Returns `{error, need_repair}` if table is corrupt

ETS and DETS Tables

- ▶ Most DETS query/insert functions are identical to ETS functions
 - A few additional functions for working with the files
 - Could use a variable to make code generic

```
StorageType = ets.  
StorageType:insert(Tab, {user, donnie, ...}).
```
- ▶ Functions to migrate between ETS and DETS
 - `dets:from_ets(Name, EtsTab)`
 - `ets:to_dets(Tab, DetsTab)`
 - Move records from an ETS table into a DETS table
 - `dets:to_ets(Name, EtsTab)`
 - `ets:from_dets(Tab, DetsTab)`
 - Move records from a DETS table into an ETS table

What Else?

- ▶ Didn't talk about one other major storage mechanism!
- ▶ Mnesia:
 - A distributed database system written in Erlang
 - Designed for telecommunications systems
 - Provides soft-realtime guarantees for queries
 - Looking up records by key *usually* completes within a very short, fixed time interval
 - Also has nested, distributed transaction support
 - (Name is a play on “amnesia,” something that you normally try to avoid with databases...)
- ▶ Definitely worth looking into!

References

▶ Bit syntax

- Erlang [Programming Examples](#), chapter 4
 - Lots of tips, includes IPv4 packet example
- [Programming Erlang](#) by Joe Armstrong, chapter 13
 - Shows how to parse ID3 tags from MP3 files

▶ ETS/DETS tables

- [Programming Erlang](#) by Joe Armstrong, chapter 15
 - A thorough description of ETS and DETS tables
- Erlang [ERTS Users Guide](#), chapter 1
 - Detailed specification of match expressions
- Erlang [Efficiency Guide](#), chapter 4
 - Guidelines for improving ETS table performance