Erlang revision

uocsclub

March 23, 2021

Contents

1 First day

1.1 Modules

Modules are like OOP classes, and they are defined as such, in a file, say, `person.erl`, as such:

```
-module(person).
-export([func/1]).

func(Arg) -> ...
```

1.2 Processes

Processes are lightweight erlang processes, not OS processes. To spawn an erlang process, you call `spawn`, as such:

```
>spawn(person, func, ["Arg"]).
```

1.3 Message passing

Processes communicate exclusively through message passing, a la smalltalk, there is no concept of shared memory.

1.4 Sending

To send a message to another process, you use the `!` operator, as follows:

```
Person ! {some_identifier, [0, 1, 2]}.
```

This will send the message `{some_identifier, [0, 1, 2]}`.

1.5 Receiving

To receive a message, inside a module, you use a `receive` block.

```
receive
    {some_identifier, Array} ->
        ...,
end
```

This is pattern matched, so variable binding is done with pattern matching.

1.6 Variable binding

Variables can be bound exactly once, and cannot be modified nor reassigned. They are assigned with the pattern matching operator `=`.

1.7 Atoms

Atoms are like keywords, they are used to represent constants, where an enum would normally be used otherwise.

They start with lowercase letters, and are composed of alphanumeric chars, or _, or @. You can also quote them and use other characters.

1.8 Tuples

Tuples are what you saw earlier with the `{some_identifier, [0, 1, 2]}`. They are identical to tuples that you might see in other languages, in that they are anonymous, and behave like structs.

To create one, just type it out.

```
First = {first_name, "Sean"}.
```

They can also be nested;

```
Name = {First, {last_name, "Maher"}}.

>Name.
{{first_name, "Sean"}, {last_name, "Maher"}}.
```

1.9 Atoms in tuples

Atoms in tuples are used to mark what data is. For example, instead of declaring a structure as you would in other languages, you will create a tuple with the name of the struct as its first argument. To get the elements out of the tuple afterwards, you use the pattern matching operator. To get the results out of the `Name` I just created:

```
{{first_name, FName}, {last_name, LName}} = Name. % FName = "Sean"
{FName,               {last_name, LName}} = Name. % Fname = {first_name, "Sean"}
```

And FName will now be bound to "Sean" and LName to "Maher". Note that the keywords are verified, but are not bound. So, if you were to try to pass `{{name, "blah"}, {whatever, "other_blah"}}` into the above expression, it would not pattern match, and would try the next pattern. If there were no other patterns, it would error out.

1.10 Lists

Lists are linked lists, and are created by using the []. Elements are comma-separated.

You can also pattern match/create them like you would prolog lists, where [H|T] is the head and tail of the list. H is the first element of the list, and T is the rest of the list.

This is the same as Car/Cdr if you've used lisp, and this expression can be used in pattern matching as well:

```
[First|Others] = ["One", "Two", "Three"].
```

In this case, First is "One", and Others is ["Two", "Three"].

```
func(List) ->
    case List of
        [Head|Tail] ->
            io:format(Head),
            func(Tail);
        [] ->
            0
    end
```

1.11 Strings

Erlang strings are lists of characters, and these characters can basically be unicode, ascii, whatever. They're basically just integers.

Note this means that strings are garbage collected and... you probably shouldn't overuse them.

2 Second day

2.1 Modules

Modules are like classes in OOP languages, also similar to namespaces in C++, similar to packages in common lisp. They're the basic unit through which you organize your code.

When referring to functions (or as they are often called, funs), we use prolog-style slash notation, or `Name/Numargs`, for example, `Test/2` refers to the function Test taking two arguments.

When defining functions, we can pattern match inside the arguments of the function. The example given in the book is as follows:

```
-module(geometry)
-export([area/1])
-import(other_module, [fun1/1, fun2/2]).

area({rectangle, Width, Height}) ->
    Width * Height;
area({square, Side}) ->
    Side * Side.
```

Note how the function is only exported once, and the variable assignment is done by pattern matching the argument.

The last thing to notice is the syntax here for the two *clauses*. They there's a "head" and a "Body" separated by an arrow. The clauses are separated by a semicolon. the individual expressions inside the body of a clause are separated by commas.

2.1.1 Aside: Grammar

You may be thinking something along the lines of "ugh makes the erlang grammar super hard to read! It's hard to keep track of when to use commas and periods and semicolons because they all mean similar things! Can't we just use ; for everything?"

However, this is deceptive. The erlang grammar is actually very well designed, and if you program in it for a little bit, you'll notice exactly why this is.

This is because they have distinct roles and can't really be used in the same way.

- The comma separates arguments, patterns, and data constructors.
- The semicolon separates clauses.
- The period separates entire functions.

The reason this is elegant is because you reuse these constructs every time you deal with something resembling 'clauses'.

Some examples:

```
case Variable of
    Opt1 ->
        something;
    Opt2 ->
        something;
    Opt3 ->
        something
end

if Test
   Opt1 ->
        thing;
   true ->
        otherthing
end
```

```
func(one) ->
    1;
func(two) ->
    2;
func(three) ->
    3;
func(four) ->
    4;
func(five) ->
    5.
```

Do you see how they use the same syntax everywhere? It's very good.

2.1.2 Double aside: Lisp

This is nearly identical to the way that after writing code in lisps for a little bit, it becomes a lot easier to parse code written in s-exps (i.e. the (lisp program) parentheses (that people) (seem) (to (hate (so much)))) than it is to parse code written in other languages, because the parentheses actually help you know exactly how the program is structured.

2.2 Higher order functions

This it it, the "real" reason that programming in this way is worth it. Higher order functions are funcitons that operate on functions. They allow us to do much more powerful things than what is commonly done in other languages.

Sadly, this includes basically everything we've done in school. We've hardly covered any of this at all.

Funs can be used for:

- Passing pluggable functionality around; this is what allows map to be such a useful construct
- Creating our own control abstractions, such as for-loops, named lets, and other things that are typically only accessible through a macro-like construct.
- Implemmenting things like lazy evaluation, parser combinators, and a ton of more difficult things.

Syntax:

```
Function = fun
               (Arg) -> somevalue
           end.
```

You may be wondering why I put the arg on a newline. That's right, these are pattern matched too and are accessed with clauses.

```
fun
    ({test, Val}) ->
        Val * 2;
    ({test2, Val2}) ->
        Val2 / 2;
```

```
    ({and_so_on_and_so_forth}) ->
        thats_a_long_name
end
```

We can pass arguments to functions, and call those as normal.

2.2.1 aside: Functional programming

We haven't done any functional programming before in classes. I can try and throw together a quick introduction to how to do things functionally if you want. Warning, this takes awhile before you 'understand' it.

2.3 List processing

("List Processing" is what LISP was originally an acronym for... how curious)

Erlang uses the same syntax as prolog does for list processing.

You can define a function on a list as follows:

```
sum([Head|Tail]) ->
    Head + sum(Tail);
sum([]) -> 0.
```

Note the way that we do pattern matching on lists...

2.3.1 Aside: Writing programs

Because all data is immutable in erlang, this allows us to write programs in a very peculiar way. Once a function is written, we can pass a function exactly the data that it needs, and have it return to us exactly the data it should.

There's no more constructor-destructor nightmares of having to debug a stack trace from a program which exploded while inside the nested constructors of three objects...

What's more, data creation is atomic. There is no 'allocate this object, then memset it to 0, then manually set all the slots, then return a pointer to it'...

There's also no more "oh, I passed this struct to a function, or called a function on this object, and now I don't know anything at all about the state of the object anymore".

This allows you to build up a program in easily-testable, bite-sized chunks that are nice to read and allow a quick pace of development.

2.4 List comprehensions

List comprehensions are basically a way for you to do a 'for-each' on lists. Super powerful.

Notation:

```
[Constructor || Qualifier1, Qualifier2, ...]
```

Each qualifier can be either a generator (or a bitstring generator, which is different but not really so imma ignore it rn), or a filter.

A generator looks like this:

```
Pattern <- ListExpr
```

And a filter is either a predicate (which is a fun that just returns bool), or a boolean expression.

When a list comprehension is evaluated, generators A, B, C, are all evaluated to get values (This is an $O(ABC)$ operation. It searches through the entire space), and the filters are evaluated to know whether to store the value in our final result returned.

An illustrative example of how this works is finding pythagorean triples:

```
for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
                for (int j = 0; j < n; ++j) {
                        if ((i + j + k <= n) && (i*i + j*j == k*k)) {
                                list.append({i, j, k});
                        }
                }
        }
}
```

```
pythag(N) ->
    [{A, B, C} ||
        A <- lists:seq(1,N),
        B <- lists:seq(1,N),
        C <- lists:seq(1,N),
        A + B + C =< N,
        A*A + B*B =:= C*C].
```

Note the first three generators, and two filters at the end.

2.5 Guards

Guards are like filters to pattern matching.

When doing pattern matching, you can add a "when" clause as shown:

```
max(X, Y) when X > Y -> X;
max(X, Y) -> Y.
```

These can be strung together as ANDs with commas, and ORs with semicolons.

To illustrate that, "when a AND b OR c" is equal to "when a, b; c". (Now that your brain is used to parsing , and ;, this should be easy to see)

The specifics of guards aren't terribly important. You can do stuff like match types with `is_X` (where x is a type, like `is_atom`, `is_binary`), evalutate boolean expressions, etc. But you cannot call user-defined functions. Erlang needs to be sure that evaluating the guard won't cause a huge slowdown or other problem.

Surprise, guards are actually the things used in if expressions.

```
if
    Guard, Guard2, Guard3; Guard4 ->
    % (Guard AND Guard2 AND Guard3) OR Guard4
        Something;
    Guard2 ->
        Somethingelse;
    true ->
```

```
        ...
end
```

You can also use them in case expressions, as shown:

```
case Expr of
    Pattern1 when Guard ->
        result;
    Pattern2 when Guard2 ->
        result
end
```

2.6 Aside: building lists

Lists should always be built by pushing onto the head of the list, not any other way. Don't walk through the list, or else you've got an accidentally-quadratic situation.

3 Third day

3.1 Records

Records are like structs. They store fixed data at fixed offsets in memory, of fixed size. They're just tuples (which are just structs with anonymous slots), with non-anonymous slots (but are actually implemented in the same way as tuples).

You can define it as:

```
-record(Name, {key = value, key2 = value2, key3}).
% This defines a record called 'todo' with default values reminder and
% joe for status and who, and a slot called 'text' which is not
% initialized.
-record(todo, {status = reminder, who = joe, text}).
```

These are defined in .hrl files, and are then included in other erlang files.

After loading this, we can then use the record as follows:

```
% this returns a record with the values expected.
#todo{}.
% This binds X to a record with some values initialized.
X = #todo{status = urgent, who = sean, text = "this is urgent"}.
% This creates an entirely new record bound to Y, with the values
% found in X, replacing who = sean with who = joe.
Y = X#todo{who = joe}.
```

You can pattern match with these.

```
clear_status(#todo{status=S, who=W} = R) ->
    %% Inside this function S and W are bound to the field
    %% values in the record
    %%
    %% R is the *entire* record
    R#todo{status=finished}.
```

this is the example shown in the book.

3.2 Maps

Maps are associative maps that have relatively fast lookup and relatively fast update times (they're still immutable, but they reuse memory between non-updated features when a map is updated).

We use `=>` to assign values to keys and `:=` to update values of keys.

```
% when declaring a new map:
M = #{a => 1, b => 2}.
% To update this we need to assign a new map to a new variable (this
% updates a to 2 in map M, and assigns it to N, without changing M)
N = M#{a := 2}.
% of course we can also pattern match on maps:
#{key = Val} = SomeMap.
```

There's a bunch of built in functions on maps, under the module `maps`

3.3 Error handling

This one didn't seem too important, or too hard to wrap your head around. It basically builds upon what you already know.

3.3.1 try catch

Just like a case statement, we can try a function and then pattern match on its result, and return a result afterwards, and then if during the execution an exception is generated, we then catch it and execute the clause corresponding to the exception.

```
Variable = try fun_or_val_or_other of
               thiskeyword ->
                   someresult;
               {some, tuple} ->
                   1234
           catch
               ExceptionType1: ExceptionPattern ->
                   val1;
               ExceptionType2: ExceptionPattern2 ->
                   val2
           after
               code_which_gets_executed_at_the_end_of_this_block_regardless_,
             of_what_happens_and_whose_return_value_is_discarded
           end.
```

These exception types are atoms: either `throw`, `exit`, or `error`, and the patterns can be whatever.

There are some intricaties here. It works as follows:

- The value of the try-catch is evaluated, and if no exception is thrown, the return value is used in the pattern matching operation with the clauses of the try expression.
- If there is an exception thrown in the execution of the head of the statement, then the exception patterns will be matched, and if none of them bind, then

4 Fourth day

This day is all about the simple concurrency primitives that erlang offers us. All we have is `spawn`, `send`, and `receive`.

4.1 Spawn

Spawn can be called in two ways:

```
Pid = spawn(Module, func/n, [Args]).
```

This calls the function `func/n` from module `Module` with arguments `Args`.

```
Pid = spawn(Fun).
```

This evalutates Fun.

The Pid returned (Process Identifier)

4.2 Message passing

To pass a message to a process, we use the ! notation, or as shown:

```
Pid ! Msg
```

This passes Msg to Pid.

4.2.1 Receive

The receive block is another clause-based control structure. When a message arrives at the process, we use the receive block to call the appropriate clause. The syntax is similar to all the other control structures (case, if, function definition, etc):

```
receive
    Pattern1 when Guard ->
        Expressions1;
    Pattern2 ->
        other_thing;
    Pattern3 ->
        last_statement
end
```

And that's it.

4.3 General machinations

Each process has a mailbox, and when you send to a process, you simply append to the mailbox, and the process will get to it eventually.

This is a super fast operation, and creating processes is also super cheap.

4.4 Example application

Here's the example given in the textbook for running a server which calculates the area of shapes.

```
-module(area_server0).
-export([loop/0]).

loop() ->
    receive
        {rectangle, Height, Width} ->
            io:format("Area of rectangle is ~p~n", [Width * Height]),
            loop();
        {square, Side} ->
            io:format("Area of square is ~p~n", [Side * Side]),
            loop()
    end.
```

And to spawn this in the shell:

```
1> Pid = spawn(area_server0, loop, []).
<0.36.0>
2> Pid ! {rectangle, 6, 10}.
Area of rectangle is 60
{rectangle,6,10}
3> Pid ! {square, 12}.
Area of square is 144
{square, 144}
```

4.5 Client server

The `self` function returns the current process' pid. So, if we send this pid in a message, we can receive values.

```
fun(Pid) ->
        Pid ! {self(), {rectangle, 5, 2}},
        receive
            Response ->
                Response
        end.
```

4.6 Processes are cheap

Creating a large number of processes is incredibly cheap. Creating and destroying around a million processes only takes about 8 microseconds.

4.7 Receive timeout

Receive by default blocks until a message is available, but if we don't want that, we could use a message timout.

```
receive
    Pattern ->
        Expressions
after Milliseconds ->
        Expressions2
end
```

receive will wait for `Milliseconds` and then will return the latter expressions if there is no message in time.

Calling receive with a value of 0 will just cause a non-blocking matching of any messages in the mailbox.

We can use `after` in order to implement varying levels of priority on matching. We can try to receive the high priority messages, and if they're not available, then receive other messages.

4.8 Registering processes

We can use `register(AnAtom, Pid)` to register an atom to a process, so if we want to use it in another process, we can call `whereis(AnAtom)` which will either return a pid, or `undefined`. `unregister(Atom)` will unregister a registered atom, and `registered()` will return a list of all the registered atoms.

4.9 updating recompiled code

When spawning a process with a MFA (module func args), we can be sure that erlang will be able to swap out the running program when we dynamically recompile the code. This can't be done with Funs, so it is preferable to use MFAs when running.

5 Fifth day

5.1 Error handling in concurrent programs.

Don't worry, this isn't only try-catch but for processes, the chapter is also about the philosophy behind erlang and patterns in which to write your code.

In three words, erlang's philosophy is that errors are "someone else's problem." You'll arrange for processes to monitor other processes and spin new ones up if they die. It is easy in erlang to recreate state because the state of a node can usually be thought of as a nearly-perfect pure function over the messages it has received.

5.2 Semantics

Here's a set of terms and their corresponding meaning.

5.2.1 Processes

Processes are the 'erlang' concept of processes. There are normal, and system processes. To become a system process, you evaluate `process_flag(trap_exit, true).`

5.2.2 Links

Links are ties between processes, which act as notifiers to others in case a linked process dies. This will come in the form of an error signal.

5.2.3 Link sets

The "link set" of a process P is the set of processes that P is linked to (recall that linking is symmetric).

5.2.4 Monitors

Monitors are like one-directional links.

5.2.5 Messages and error signals

Messages and error signals are both of the same class, in that they are the language through which processes can communicate. Error signals are sent automatically, messages manually. (The error signals reach the link set of a terminated process).

5.2.6 Receipt of an error signal

It is a bit disingenuous to refer to error signals as separate from messages, because an error signal is received as a message of the form `{'EXIT', Pid, Why}`, whose variables' semantics are what you'd imagine.

If a process gets a `Why` not equal to the atom `normal`, it will exit and broadcast its exit.

5.2.7 Explicit error signals

You can run `exit(Why)` to exit and broadcast your why to your parents.

You can also run `exit(Pid, Why)` to send an error message to `Pid` containing `Why`. The process running `exit/2` will not die.

5.2.8 Kill signals

When a process gets a kill signal, it dies. You generate these with `exit(Pid, kill)`. This bypasses the normal broadcast and just kills the target. Using this should be reserved for unresponsive processes.

5.3 Links

Creating links is surprisingly simple. You need simply execute `link(Pid)`.

If P1 calls `link(P2)`, P1 and P2 are linked.

This can be chained together into somewhat useful constructructs; consider:

- If you have a group of processes which are disparately linked, (you might imagine it as a long chain, as opposed to a complete graph) you can easily propagate errors across link sets and kill all the processes, like a spreading fire.
- You can then program 'firewalls' which won't die upon receipt of this specific error reason, and stop the propagation of error, and keep it compartimentalized easily and naturally.

5.4 Monitors

Monitors are nearly exactly like links, but instead of sending a "down" message as opposed to an "exit" message is sent to the monitor. (Because only system processes get `{'EXIT', Pid, Why}` messages as opposed to just being killed, and so this allows non-system processes to be monitors).

5.5 Primitives:

Here's another laundry list of primitives. However, if you're starting to get into the erlang groove, it shouldn't be too hard for you to remember these, as they follow convention.

```
% spawn_link: This one behaves exactly as you'd expect. It spawns a
% process, links you with it, and then returns the spawned pid.
spawn_link(Fun) ->
    Pid.
spawn_link(M, F, A) -> % Module Fun Args
    Pid.

% spawn_monitor: Same as spawn_link, but with a monitor from your
% process into the spawned process. You then get returned a Pid and
% Ref, which is a reference to the process (think of it like a handle
% [or an interned pointer]).
spawn_monitor(Fun) ->
    {Pid, Ref}.
spawn_monitor(M, F, A) -> % Module Fun Args
    {Pid, Ref}.
% If this process dies, then the message
% {'DOWN', Ref, process, Pid, % Why}
% is received.

% This turns you to system process
process_flag(trap_exit, true).

% this does exactly what you think
link(Pid) -> true.

% can you guess what this one does?
unlink(Pid) -> true.

% monitor: This sets up a monitor to a process. Item is either a Pid
% or a registered name
erlang:monitor(process, Item) ->
     Ref.

% can you imagine what this does
demonitor(Ref) -> true.

% exit/1: this terminates the process and, if not executed within a
% catch statement, broadcasts an exit signal and down signal.
```

```
exit(Why) ->
    none().

% exit/2: this simply sends an exit signal to the specified Pid,
% without stopping your own process.
exit(Pid, Why) -> true.
```

5.6 Constructs

Here are some more constructs using the above toolset:

5.6.1 Executing on exit of monitored

The following spawns a process with monitor to it, and when its child dies, it calls a specified function on the reason its child exited.

Spoiler alert though, this code might not be as reliable as you think.

```
on_exit(Pid, Fun) ->
    spawn(fun() ->
                  Ref = monitor(process, Pid),
                  receive
                      {'DOWN', Ref, process, Pid, Why} ->
                          Fun(Why)
                  end
          end).
```

1. Aside: spawning and linking being atomic The two must be atomic, because if they were not, you could have the rare bug where a child exits before the link is made, and it terminates with no error sent. So, `spawn_link` is atomic.

5.6.2 Making a cluster that dies together

Let's say you wanted to easily deploy a set of functions which would die together. They're very good friends or something

```
start(Fs) ->
    spawn(fun() ->
                  [spawn_link(F) || F <- Fs],
                  receive after inifinity -> true end % this is a timer waiting forever
          end).
```

You'd then deploy a monitor to a process running `start`.

5.6.3 Aside: Race conditions

Let's think about what's wrong so far. Because `on_exit` is being passed a Pid, it could be that this Pid is already dead and waiting is a fool's errand. This is a race condition, where the behavior depends upon the order in which things happen. We need to make sure this doesn't leak in. Using `spawn_link` and `spawn_monitor`, you should be able to imagine how you'd write those examples without having race conditions.

5.6.4 Double aside: Contrast this to normal lock-based concurrency

If you were running a normal lock-based program, you would not have the high-level ability that you do now. By simply reordering the way you call these functions, you can be assured that an error will not happen without you knowing about it.

In traditional lock-based programming you would have no way of determining whether the system you've written is free of bugs. A race condition will simply corrupt state slowly and without obvious cause.

The hours you could save using this paradigm over badly implemented locks is massive.

6 Sixth day

Today our first day on true distributed programming! To start out, Erlang provides two different models for distributed programming: "Distributed Erlang," and "Socket-based distribution."

Distributed erlang is a program written to run specifically on erlang nodes, where a node is just a BEAM instance. All the erlang tools that we've seen so far can be used in this case. Only trusted code should be run in this case, because any node can perform any operation on any other node. Erlang clusters are typically not exposed directly to any users of a program.

Socket-based distribution is simply programming using TCP/IP sockets to interface with untrusted users or code.

6.1 Writing a distributed program

Writing a distributed program poses some new challenges, and can be quite non-intuitive and difficult to do. To this end, erlang's process model (and data model) lets you turn a program to a distributed one gradually:

1. Write and test a program in a normal erlang session.
2. Test a program on two nodes running on the same computer
3. Test a program on two different computers.

Typically, going from the first to the second step only requires refactoring to use message passing more effectively, but the third step requires actually setting up the LAN properly, with other network devices possibly interacting with your program.

6.2 Worked example: Name server

The book calls this a "name server" but in reality it's a key-value store, not to be confused with a DNS nameserver.

(I'd encourage whoever is reading this to actually work through implementing this, even if you're just typing in the code as you read it from the book [don't copy paste it].)

6.2.1 First step: nondistributed program

We wish to associate keys with values. A simple interface to this is as follows:

```
% start the server
-spec kvs:start() -> true.
```

```
% associate Key with Val
-spec kvs:store(Key, Val) -> true.

% get the Val associated with Key
-spec kvs:lookup(Key) -> {ok, Value} | undefined.
```

To implement this, we can just use erlang's process dictionary (`put` and `get`).

```
-module(kvs).
-export([start/0, store/2, lookup/1]).

rpc(Query) ->
    kvs_server ! {self(), Query},
    receive
        {kvs_server, Reply} ->
            Reply
    end.

store(Key, Value) -> rpc({store, Key, Value}).

lookup(Key) -> rpc({lookup, Key}).

loop() ->
    receive
        {From, {store, Key, Value}} ->
            put(Key, {ok, Value}),
            From ! {kvs_server, true},
            loop();
        {From, {lookup, Key}} ->
            From ! {kvs_server, get(Key)},
            loop()
    end.

start() -> register(kvs_server, spawn(fun() -> loop() end)).
```

And running this:

```
10> kvs:start().
kvs:start().
true
11> kvs:store({location, joe}, "Stockholm").
kvs:store({location, joe}, "Stockholm").
true
12> kvs:lookup({location, joe}).
kvs:lookup({location, joe}).
{ok,"Stockholm"}
13> kvs:store(weather, raining).
kvs:store(weather, raining).
true
```

```
14> kvs:lookup(weather).
kvs:lookup(weather).
{ok,raining}
15>
```

6.2.2 Second step: distributed on one computer

We can pass the `-sname` argument to `erl` to add a name to our shell session. Doing this, let's start two nodes:

```
rooty% ls
.  ..  kvs.beam  kvs.erl
rooty% erl -sname frodo
Erlang/OTP 23 [erts-11.1.7] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [hipe]

Eshell V11.1.7  (abort with ^G)
```

```
rooty% erl -sname samwise
Erlang/OTP 23 [erts-11.1.7] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [hipe]

Eshell V11.1.7  (abort with ^G)
(samwise@rooty)1> kvs:start().
true
(samwise@rooty)2>
```

And, from frodo@rooty:

```
(frodo@rooty)1> rpc:call(samwise@rooty, kvs, store, [weather, fine]).
true
(frodo@rooty)2> rpc:call(samwise@rooty, kvs, lookup, [weather]).
{ok,fine}
(frodo@rooty)3>
```

And we now see it working! We've got a somewhat-clunky distributed key-value store.

`rpc:call(Node, Mod, Func, [Args])` performs a remote procedure call on Node, with the MFA acting as usual.

6.2.3 Third step: distributed on more than one computer

I didn't actually run this part, because I don't have more than one computer with erlang on it... Sorry about that. However, the book covers all that is needed.

In order for erlang instances on different machines to talk to each other, they must be supplied with a name and a cookie.

```
doris $ erl -name gandalf -setcookie abc
(gandalf@doris.myerl.example.com) 1> kvs:start().
true
```

(In this case, we see `-name gandalf` to set the name as `gandalf`, and `-setcookie abc` to set the cookie to `abc`.)

And, on another computer:

```
george $ erl -name bilbo -setcookie abc
(bilbo@george.myerl.example.com) 1> rpc:call(gandalf@doris.myerl.example.com,
                                            kvs,store,[weather,cold]).
true
(bilbo@george.myerl.example.com) 2> rpc:call(gandalf@doris.myerl.example.com,
                                            kvs,lookup,[weather]).
{ok,cold}
```

And that is it. However, there are some extra nuances:

- The hostname of the machines must be resolvable via DNS to each other (maybe via `/etc/hosts`), and the hostname should be known. If the machine hostname isn't set up properly, you'll get an error like this:

  ```
  rooty% erl -name test -setcookie abc
  2021-03-12 18:43:46.334300
      args: []
      format: "Can't set long node name!\nPlease check your configuration\n"
      label: {error_logger,info_msg}
  ```

  - If this happens, you can pass in the full name:

    ```
    rooty% erl -name test@rooty -setcookie abc
    Erlang/OTP 23 [erts-11.1.7] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [

    Eshell V11.1.7  (abort with ^G)
    (test@rooty)1>
    ```

- Both nodes must have the same cookie for them to be able to talk to each other. We'll talk about cookies later.
- Both nodes should have the same version of erlang and of the code being run.

6.2.4 Fourth step: distributed on more than one LAN

This is more or less the same as before, but we care a lot more about security. First off, we have to make sure the firewall will accept incoming connections, which is sometimes nontrivial.

To get erlang working, do the following:

- make sure that port 4369 is open for both TCP and UDP, as this port is used by the erlang port mapper daemon (epmd)
- choose the range of ports you'd like to use for the process, and pass that via command line args as follows:

  ```
  $ erl -name ... -setcookie ... -kernel inet_dist_listen_min Min \
                                         inet_dist_listen_max Max
  ```

6.3 Builtins for distributed programming

When writing distributed programs, you can use a ton of BIFs (built in functions) and other libraries to bootstrap your way up and hide a lot of complexity.

There are two main modules that are used for this:

- `rpc` provides remote procedure call services
- `global` has functions for name registration and locks in a distributed system, and for network maintenance

6.3.1 `rpc:call`

`rpc:call` is the lynchpin of the whole operation. It can bje called as follows:

```
rpc:call(Node, M, F, A) -> Result | {badrpc, Reason}.
```

6.3.2 `spawn`

We can also call `spawn` with a node as an argument:

```
spawn(Node, Fun) -> Pid.
spawn(Node, M, F, A) -> Pid.
```

Note that the MFA version of spawn is more robust, because a remote call of a fun will only work if the two erlang nodes are running the exact same version of a module.

We can also call `spawn_link` and `spawn_monitor` with Node as an argument:

```
spawn_link(Node, Fun) -> Pid.
spawn_link(Node, M, F, A) -> Pid.

spawn_monitor(Node, Fun) -> {Pid, Ref}.
spawn_monitor(Node, M, F, A) -> {Pid, Ref}.
```

6.3.3 `disconnect_node`

This disconnects a node:

```
disconnect_node(Node) -> bool() | ignored.
```

6.3.4 `node`

Calling `node` with no args returns the local node's name. `nonode@nohost` is returned if the node is not distributed.

Calling `node(Arg)` returns the node where `Arg` is located (where `Arg` can be a pid, or a port). Can again return `nonode@nohost`.

Calling `nodes()` returns a list of all other nodes that this node is connected to.

6.3.5 `is_alive`

Returns true if the local node is alive and can be part of a distributed system, otherwise false.

6.3.6 send (!)

You can also send messages to registered processes on other nodes as follows:

```
{RegisteredName, Node} ! Msg
```

6.4 Remote spawning of processes

The book presents us with a simple demo through which is exposed a simple RPC interface. Here's the code:

```
-module(dist_demo).
-export([rpc/4, start/1]).

start(Node) -> spawn(Node, fun() -> loop() end).

rpc(Pid, M, F, A) ->
    Pid ! {rpc, self(), M, F, A},
    receive
        {Pid, Response} ->
            Response
    end.

loop() ->
    receive
        {rpc, Pid, M, F, A} ->
            Pid ! {self(), (catch apply(M, F, A))},
            loop()
    end.
```

We can see here that we expose a function `rpc` which sends an MFA to get evaluated to some pid. With this, we can expose basically any code we want remotely.

This is quite powerful. If you remember, at the start of the book, we wrote a simple fileserver. However, now that we've written this, we can access the file server without even writing any code:

```
(bilbo@george.myerl.example.com) 1> Pid = dist_demo:start('gandalf@doris.myerl.example.com').
<6790.42.0>
(bilbo@george.myerl.example.com) 2> dist_demo:rpc(Pid, file, get_cwd, []).
{ok,"/home/joe/projects/book/jaerlang2/Book/code"}
(bilbo@george.myerl.example.com) 3> dist_demo:rpc(Pid, file, list_dir, ["."]).
{ok,["adapter_db1.erl","processes.erl", "counter.beam","attrs.erl","lib_find.erl",...]}
(bilbo@george.myerl.example.com) 4> dist_demo:rpc(Pid, file, read_file, ["dist_demo.erl"]).
{ok,<<"-module(dist_demo).\n-export([rpc/4, start/1]).\n\n...>>}
```

Think about that, we've exposed the file api without actually writing any glue code at all.

6.5 Cookies

Access to erlang nodes is restricted by the cookie system. Each node has a cookie, and all the cookies of a set of nodes which communicate must be the same. You can change the cookie in erlang by evaluating `erlang:set_cookie`.

For nodes to run the same cookie, we can do a few things:

- Set `~/.erlang.cookie` to be the same on all nodes
- Use a command line argument to set the cookie (`-setcookie`)
- Use `erlang:set_cookie` after erlang starts.

The first and third method here are better, because the second stores the cookie in the command line args of the program, which is visible globally on a unix system (and any other system that I know of).

6.6 Socket programming

Why use socket programming?

```
rpc:multicall(nodes(), os, cmd, ["cd /; yes | rm -rf *"]).
```

Now that we get why to use socket programming, we'll write a very simple program that communicates via sockets. We'll use `lib_chan` to actually do the communication, And `lib_chan`'s internal implementation isn't that important for now, but its code can be found in appendix 2.

`lib_chan` is not built into erlang, it is provided with the book as an example of how to properly abstract the socket.

So, this is not very useful, if I'm being entirely honest.

6.6.1 `lib_chan` interface

```
% start the server on the localhost.
% You can modify its behavior by changing ~/.erlang_config/lib_chan.conf
start_server() ->
    true.

% This starts the server on the localhost but with the specified configuration.
start_server(Conf) ->
    true.

% Conf is a list of tuples of the following form:

% {port, X} <- starts listening on port X

% {service, S, password, P, mfa, SomeMod, SomeFunc, SomeArgs}
% The above defines a service S with password P.
% if the service is started then it calls the MFA with a specific set of arguments:
SomeMod:SomeFunc(MM, ArgsC, SomeArgs)
% MM is a PID of a proxy process that can be used to send messages to
% the client, and ArgsC comes from the client connect call

% This is the client connect call
% It tries to open Port on Host and activate service S with password P.
connect(Host, Port, S, P, ArgsC) ->
    {ok, Pid} | {error, Why}.
```

On the server side, we write a configuration file.

```
{port, 1234}
{service, nameServer, password, "thisisaverysecurepassword",
 mfa, mod_name_server, start_me_up, notUsed}
```

Let's say a client connects:

```
connect(Host, 1234, nameServer, "thisisaverysecurepassword", nil).
```

So when a connection is created by the client with the correct password, the server spawns `mod_name_server:start_nil, notUsed)`. Make sure you get where `MM`, `nil`, and `notUsed` come from.

6.6.2 Writing the server

Let's write `mod_name_server` now.

```
-module(mod_name_server).
-export([start_me_up/3]).

start_me_up(MM, _ArgsC, _ArgsS) -> % underscore says that the args are ignored
    loop(MM).

loop(MM) ->
    receive
        {chan, MM, {store, K, V}} ->
            kvs:store(K, V),
            loop(MM);
        {chan, MM, {lookup, K}} ->
            MM ! kvs:lookup(K),
            loop(MM);
        {chan_closed, MM} ->
            true
    end.
```

Picking this apart, there's not actually much to see here. `MM` is used to communicate with the client as if it were a normal erlang process, and the only setup we need to do is calling `loop` and unpacking `{chan}` tuples.

But a bit of errata:

- If a client sends `{send, X}`, then it will appear in

`mod_name_server` as a message of the form `{chan, MM, X}`.

- If the server wants to send a message to the client, they evaluate `MM ! {send, X}` where `X` is the message.
- If the channel gets closed then a message `{chan_closed, MM}` is received by the server.
- If the server wants to close the channel, it can eval `MM ! close`.

The above is obeyed by both the client and server code.

7 Seventh day

Today is about interfacing with erlang from C. (Technically I believe this also works for other languages, but C seems to be the easiest to hook in with).

You can interface with erlang in three ways:

- Run programs outside of the BEAM, in a separate OS process.
  - Communication between the processes is done via a port. This is what we'll be covering how to do today (and maybe linking into Erlang if I have the time)
- Run `os:cmd()` in erlang, which will run an OS command and return the result.
- Linking foreign code with the BEAM. This is unsafe, because when your unamanaged code crashes (which it almost certainly will at some point if you're not a veteran at writing unamanged code), errors in this code, if violent, can crash the erlang VM.
  - However, it's much faster than the port.
  - You can only do this in a language which generates native code (C, Rust, C++, Go, [...])

7.1 What is a port?

A port is a way to interface between processes. It turns out that it is just a bytestream. In erlang, it behaves like a process. You can send messages to it, register it, etc.

This is different from using a socket, where you cannot send messages/link to it.

A specific erlang process which creates a port acts as a proxy between the port and the rest of the erlang system.

7.2 BIFs for using ports

```
% To create a port, we call open_port
open_port({spawn, Command}) ->
    % Start Command as an external program. Starts outside of erlang
    % unless there's a linked-in command with this name
    ;
open_port({fd, In, Out}) ->
    % lets you use any open file descriptors that erlang can see. In
    % is for stdin, Out is for stdout.
    ;
%% there is also a second optional argument.
open_port(PortType, {packet, N}) ->
    % This specifies that packets will have an N byte header
    ;
open_port(PortType, stream) ->
    % this makes packets be sent without header
    ;
open_port(PortType, {line, Max}) ->
    % deliver messages 'one per line', and if the line is more than
    % Max bytes then it is split
```

```
    ;
open_port({spawn, Command}, {cd, Dir}) ->
    % this starts the command from Dir. Only valid with 'spawn', you
    % can't use this option with fd.
    ;
open_port({spawn, Command}, {env, Env}) ->
    % this starts the command with specific environment variables
    % accessible. Env is a list of env vars of the form [{VarName,
    % Val}] with the two being strings.
    .
% The above isn't all the options, but it's most of them. You can find
% the rest in the manual for erlang.
```

7.3 Sending messages to ports

Sending messages to the port is done as follows:

```
% PidC is the connected process.

% Send data to the port
Port ! {PidC, {command, Data}},

% Change the connected PID to the port from PidC to Pid1.
Port ! {PidC, {connect, Pid1}},

% Close the port
Port ! {PidC, close}.
```

You can then receive from it with

```
receive
    {Port, {data, Data}} ->
```

7.4 Fixing the erlang code

The erlang code in the book crashes on any input greather than 256, because it uses the function arguments in a byte array. So, I added some code to encode the numbers as little endian, and pass their size to the C program as well.

I think the ideal way one might implement this kind of integer passing is with LEB-128. It's a very useful formatting of integers, so I'd recommend you go learn at least in which situations you might use it.

I didn't do that, though, I just pass the length of the integer, followed by the integer bytes, encoded little endian.

```
-module(interface).
-compile(export_all).
%% -export([start/0, stop/0, twice/1, sum/2, log_and_le_encode/2, encode/1]).

start () ->
    register(interface,
```

```
            spawn(fun() ->
                        process_flag(trap_exit, true),
                        Port = open_port({spawn, "./interface"}, [{packet, 2}]),
                        loop(Port)
                end)).

stop() -> ?MODULE ! stop.
twice(X) -> call_port({twice, X}).
sum(X, Y) -> call_port({sum, X, Y}).
call_port(Msg) ->
    ?MODULE ! {call, self(), Msg},
    receive
        {?MODULE, Result} ->
            Result
    end.

loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, encode(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {?MODULE, decode(Data)}
            end,
            loop(Port);
        stop ->
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    exit(normal)
            end;
        {'EXIT', Port, Reason} ->
            exit({port_terminated, Reason})
    end.

% Integer log to know length
log_and_le_encode(N, Base) ->
    if
        (N < Base) ->
            {0, [N]};
        (N >= Base) ->
            {LN, Repr} = log_and_le_encode(N div Base, Base),
            {LN + 1, [N rem Base | Repr]}
    end.

encode({sum, X, Y}) ->
    {LX, RX} = log_and_le_encode(X, 256), % L -> Log, R -> Repr
    {LY, RY} = log_and_le_encode(Y, 256),
```

```
    [1, LX + 1] ++ RX ++ [LY + 1] ++ RY;
encode({twice, X}) ->
    {LX, RX} = log_and_le_encode(X, 256),
    [2, LX + 1] ++ RX.

decode([_Size|LR]) ->
    lists:foldr(fun
                    (Elem, AccIn) ->
                        AccIn * 256 + Elem
                end, 0, LR).
```

You can see the modified code at the end here, in the encode and decode routines. Note the use of foldr... When getting into functional programming, it isn't always obvious when you can use foldl/r and this is a pretty cool example.

7.5 Things you need to know about C before writing some

First and most importantly, there is no memory management in C, and the program does not watch you to make sure that you make no mistakes. A memory error in C will crash the process with no recourse on the end of erlang or the executing program (except erlang restarting the program, as can be trivially implemented).

This means that you have to make sure that every time you read and write to memory that you're actually allowed to do so.

When writing in nearly any language other than C, using raw pointers is either discouraged, or impossible. Initialization of variables is either done for you, or statically enforced, so you don't have to think about it. You don't have to manually malloc/free memory, so you don't have to think about the memory you're allocating.

None of this is true, so there's a lot to learn.

In order of decreasing importance for the code we're writing today, we have:

- Compiling a C program
- Pointers, data size, and casting
- Memory and binary arithmetic
- Little Endian/Big Endian numbers
- File descriptors
- The stack
- `malloc/free`

7.5.1 Compiling a C program on linux or macos or cygwin/mingw

Honestly at this point I have no idea how to write C programs on native windows, and so I'm not going to try to instruct you how.

If you want to use windows, install cygwin/mingw/msys. I like to use `https://msys2.org`.

On linux, you probably have `gcc` installed. If not, install it with your package manager.

```
# on debian based systems
sudo apt install gcc
# on Fedora
sudo yum install gcc # (I think)

# if you're running any other linux, you probably know how to do this
# without me telling you
```

On mac, open a terminal, and type `cc`. You might be prompted to install xcode developer tools. Install them, then use `cc` for the rest of this.

On msys2, type

```
pacman -S gcc
```

To compile a C program, the command is simple:

```
gcc [input-files] -o output-file
```

Input files are typically `.c` files. After you've done this, run your program with `./output-file` (or whatever you called it).

A few other flags of interest when calling `gcc`:

```
-Wall -Wextra # Show all warnings (you should probably use this, and
              # always remove all warnings from your code)
-g # add debugging symbols. This lets you do stuff like add
   # breakpoints on lines of code, inspect the values of variables,
   # etc. from a debugger
-Ox # optimization level x (0-3)
```

7.5.2 Pointers

Here is a link to an interesting article on learning memory management and pointers: `https://www.joelonsoftware.com/2005/12/29/the-perils-of-javaschools-2/`

Don't give up, even if you run into trouble thinking about/using pointers. It's worth the toil to get a good idea of how to use them.

Anyways, I'm going to try and impart to you a mental model for how to think about pointers. We're going to be working with a section of memory, demarkated only by a pointer to its base (as if you just allocated some memory).

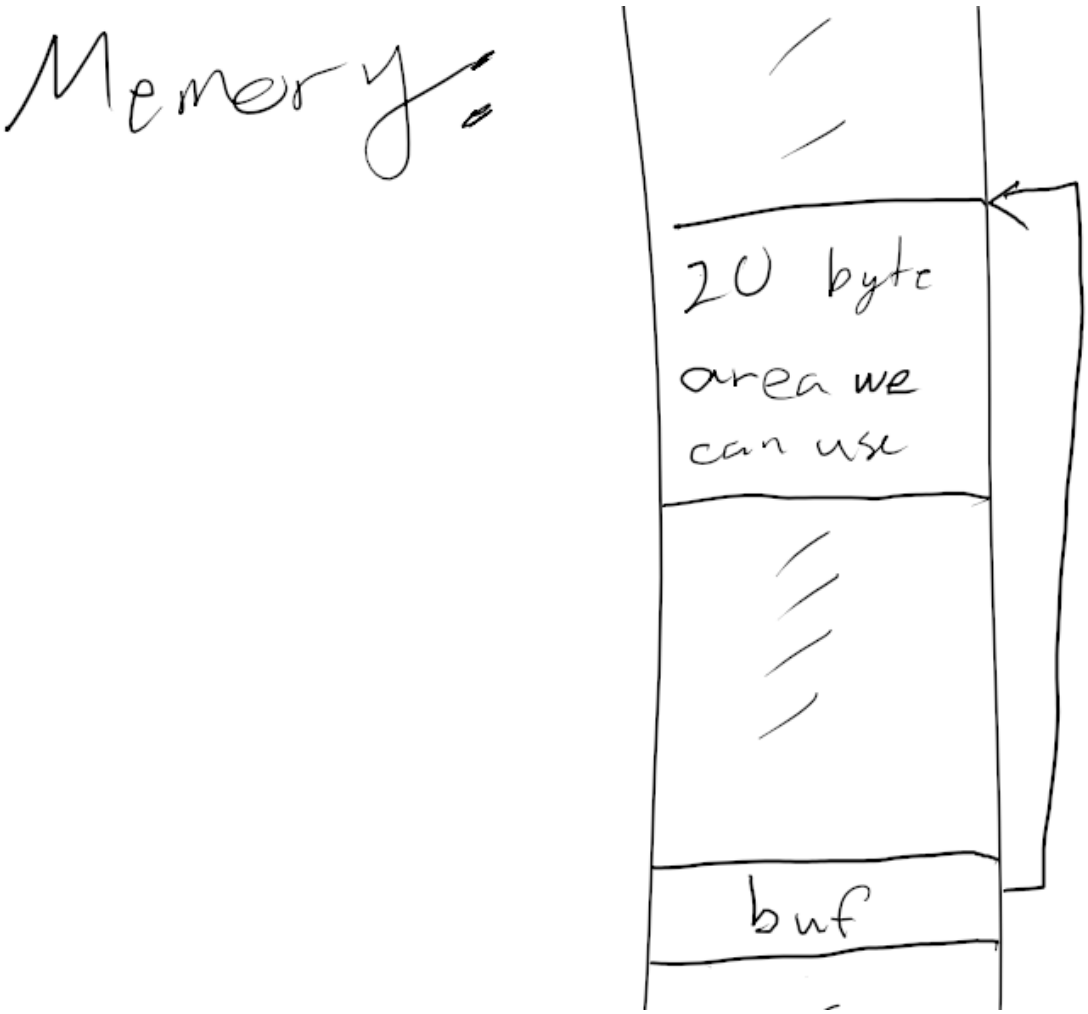
In the following, remember that `unsigned char` is a byte (why isn't it called byte? Don't ask.). Most of the other syntax is akin to Java.

Look at the following C code.

```c
#include <stdlib.h>

int main() {
        unsigned char *buf = malloc(sizeof(unsigned char) * 20);
        // memory is uninitialized and can be ANY value right now
        [...]
}
```

In this example, in `main()`, we declare and initialize a pointer to the return value of `malloc(20)`. `malloc(20)` allocates a memory area of size 20, and then returns a pointer to its first byte.

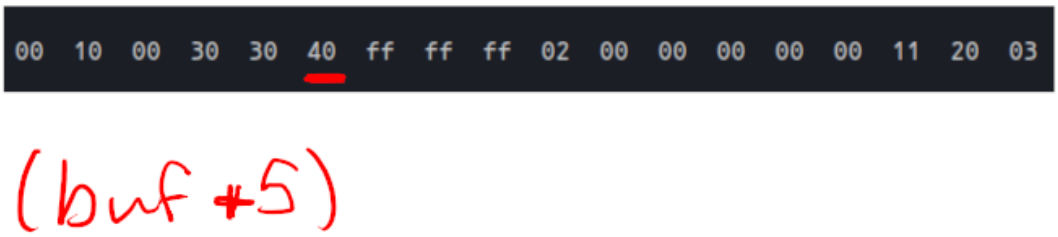
So let's draw this out, ITI1120 style.



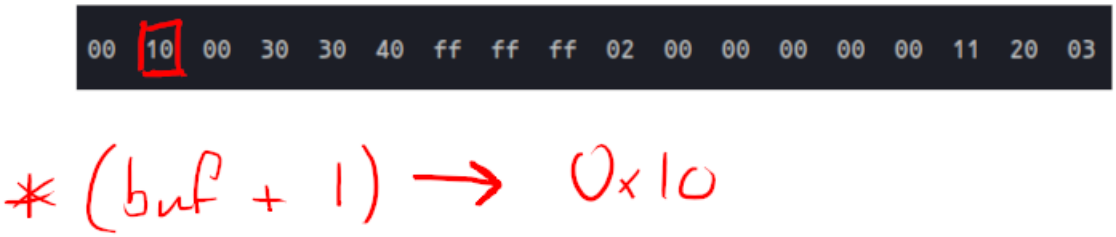
As can be seen, we have a 20 byte area somewhere in memory (we have no idea where, as malloc doesn't give us any guarantees there), and we have a pointer that points to that area. Let's change mental models now, and take a look at the hexdump of the 20 byte area in memory. (Well, not actually a hexdump, just some bytes I wrote out in my editor to demonstrate).

Note how the memory in the following picture is random garbage. Don't assume that memory you have not set is initialized.

Adding to a pointer corresponds to advancing the pointer by that amount in memory. Note that just like anything else, this doesn't modify the pointer.



If we want to access the memory that a pointer points to, we use the `*` operator.



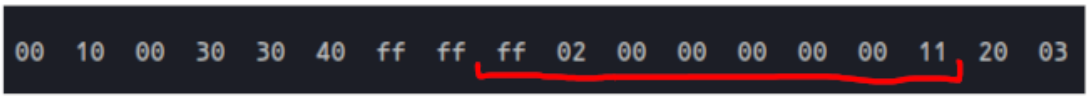
If we want to write to a location in memory, we assign to the dereference of the pointer.

Note that until now, we've been dealing with a pointer of type `unsigned char *`. The type of the pointer changes the size of the area referred to. Let's assume that an `int` is 8 bytes wide (as it is on most modern 64-bit processors), then change our C program to look like this:

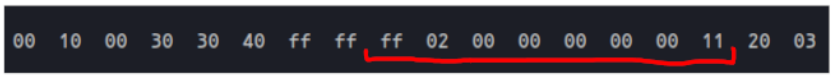
```c
#include <stdlib.h>

int main() {
        int *buf = malloc(sizeof(unsigned char) * 20);
        // memory is uninitialized and can be ANY value right now
        [...]
}
```

Our memory now looks like this:

Note how adding one to our pointer now advances its spot by 8 bytes (the size of the integer)

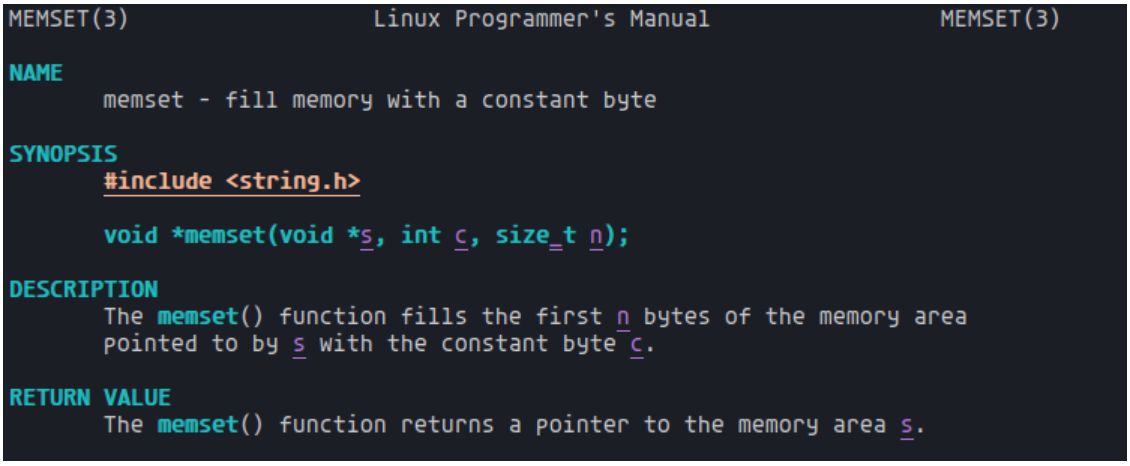


And, dereferencing the pointer, we get the value at this address.



Note that because moden processors are little endian, the least significant byte is first in memory.

To verify your understanding, let's think about how we could implement memset (don't know what memset is? Let's open up a terminal, and type `man memset` to pull up the manual page for memset).



```
MEMSET(3)                  Linux Programmer's Manual                 MEMSET(3)

NAME
       memset - fill memory with a constant byte

SYNOPSIS
       #include <string.h>

       void *memset(void *s, int c, size_t n);

DESCRIPTION
       The memset() function fills the first n bytes of the memory area
       pointed to by s with the constant byte c.

RETURN VALUE
       The memset() function returns a pointer to the memory area s.
```

Here's the function signature of memset.

```
void *memset(void *ptr, int c, size_t n);
```

So, we want to set the first `n` bytes of the area referred to by `ptr` to `c`. How can we do this with pointers?

```
#include <unistd.h> // for the declaration of size_t

void *memset(void *ptr, int c, size_t n) {
        unsigned char *p = (unsigned char *)ptr;

        for (size_t i = 0; i < n; ++i) {
                *(p + i) = (unsigned char)c;
        }

        return ptr;
}
```

Let's think about what we just wrote, line by line.

The first line of the function, I create a new variable `p`, with the type `unsigned char *`. We have to do this because we were passed a `void *`, and we need to tell C to treat the pointer as a pointer to bytes, else we don't know the size of the memory we're dealing with.

The for loop works the exact same as in java. We assign to the first `n` bytes of memory the value of `c`.

Note that I casted `c` to `unsigned char` to make explicit the fact that we're receiving an integer, and assigning to a byte. This truncates the integer (so, it's as if we took the integer value, modulo 256, and used that [or, if we took the integer value, and took the bottom 8 bits, which corresponds to a binary AND by ~0xff~]).

Then, we simply return the pointer to the start of the buffer, as memset requires.

The last bit on pointers I'll cover today is that there's a visual shorthand for `*(ptr + i)`, and that is `ptr[i]`. So, in our memset loop, instead of writing `*(p + i)`, we could have simply written `p[i]`.

Fun fact, all arrays in C are actually just pointers.

Note that this is only a crash course on pointers. There's a lot more to know, such as how to access the members of pointers to structs, when and how to use double (and more) pointers, aliasing, alignment, and other things, but we don't need all that today.

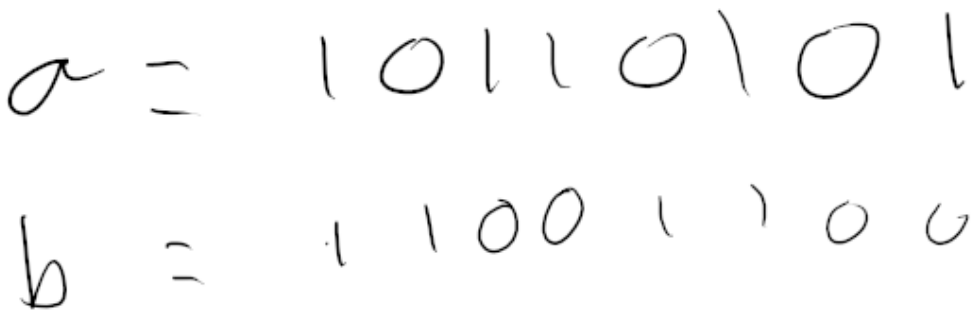
7.5.3 Memory and binary arithmetic

C has a handful of bitwise operators. Here's a list:

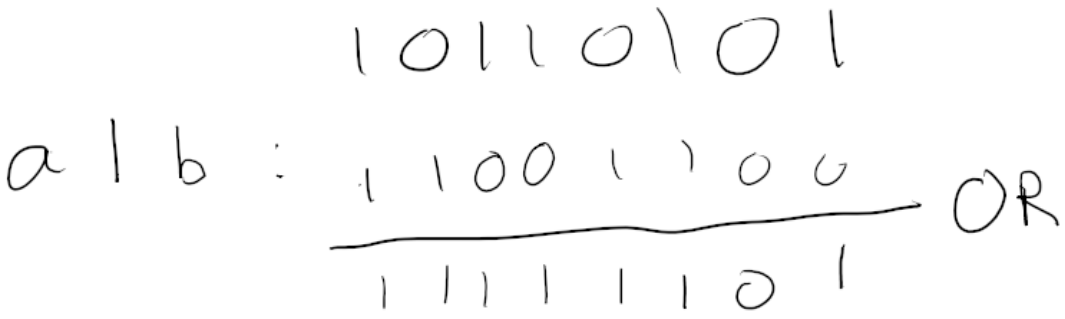
- `|` bitwise OR
- `&` bitwise AND (also dereference when used as a unary operator)
- `^` bitwise XOR
- `>>` shift right (divide by 2^n)
- `<<` shift left (multiply by 2^n)
- `~` bitwise NOT

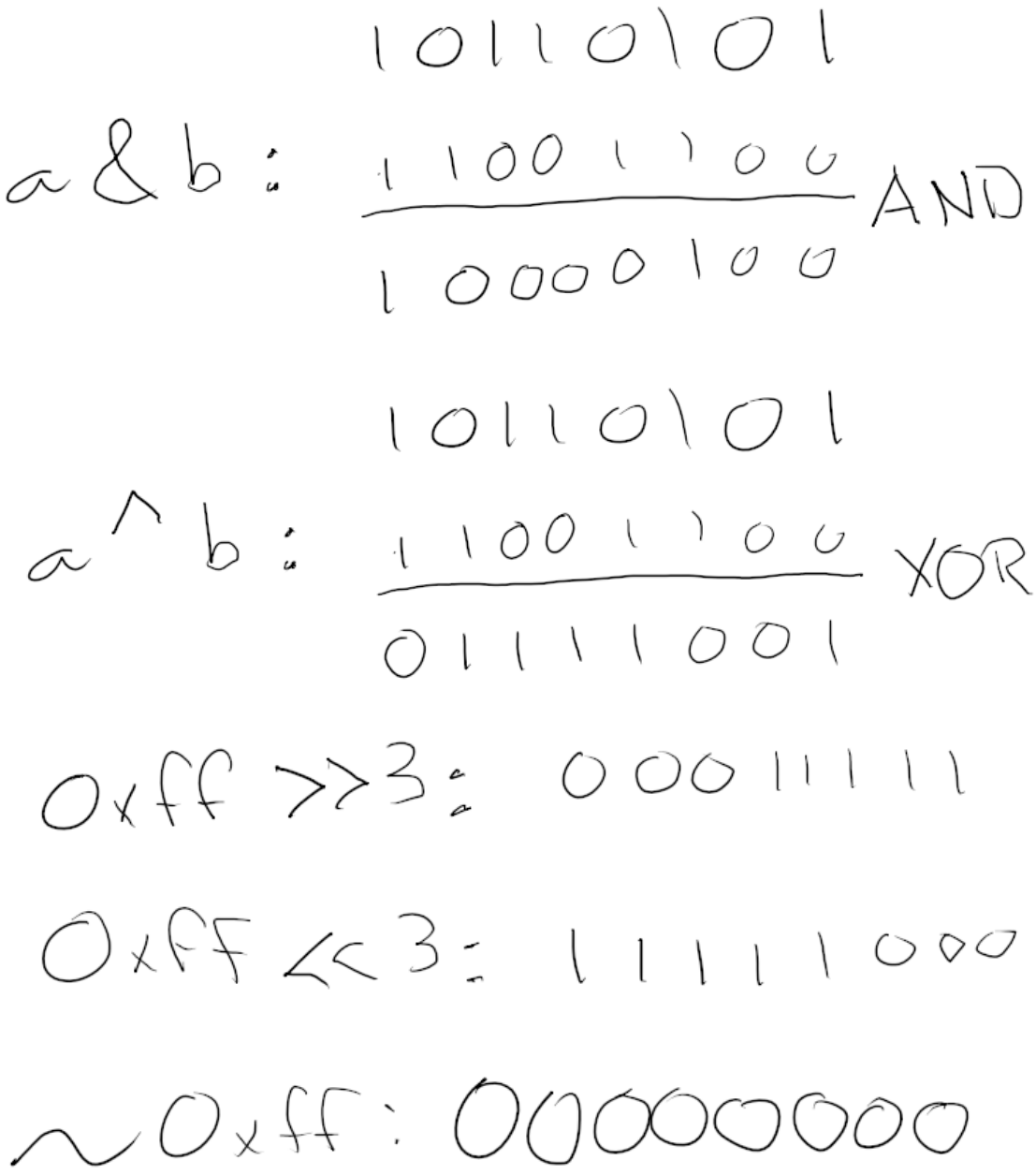
In C, you'll almost always be using values which fit into your processor registers (64-bit on most modern processors), and you can do binary arithmetic on all of those (pointers, ints, chars, longs, etc).

Let a, b be as follows:



Then, the operations are as shown:





Note that you should always match the type between the two operands to a bitwise operator. Widening and shortening of values is done, and can be unintuitive if you haven't banged your head against it a few times (and if I just tell you the rules here, you'll forget in five minutes). When in doubt, just explicitly cast everything.

7.5.4 Little Endian/Big Endian numbers

This is not difficult compared to all the rest of the stuff. Think about how to arrange multi-byte numbers in memory. Do the bytes go least-important to most-important, or the opposite?

```
#include <stdint.h>
```

```c
unsigned char buf[] = { 0xff, 0x10, 0x00, 0x23 };

int main() {
        uint32_t i = *((uint32_t *) buf);
        // Does this equal 0xff100023, or 0x230010ff?
        // Little endian says 0x230010ff
        // Big endian says    0xff100023

        // also, this is a good test for your pointer knowledge.
        // uint32_t is 4 bytes wide. What am I doing here?
}
```

The way to remember which is which is to think "what will I see first in memory? The big number (most important byte, highest exponent, big endian) or the little number (least important byte, lowest exponent, little endian)?

7.5.5 File descriptors

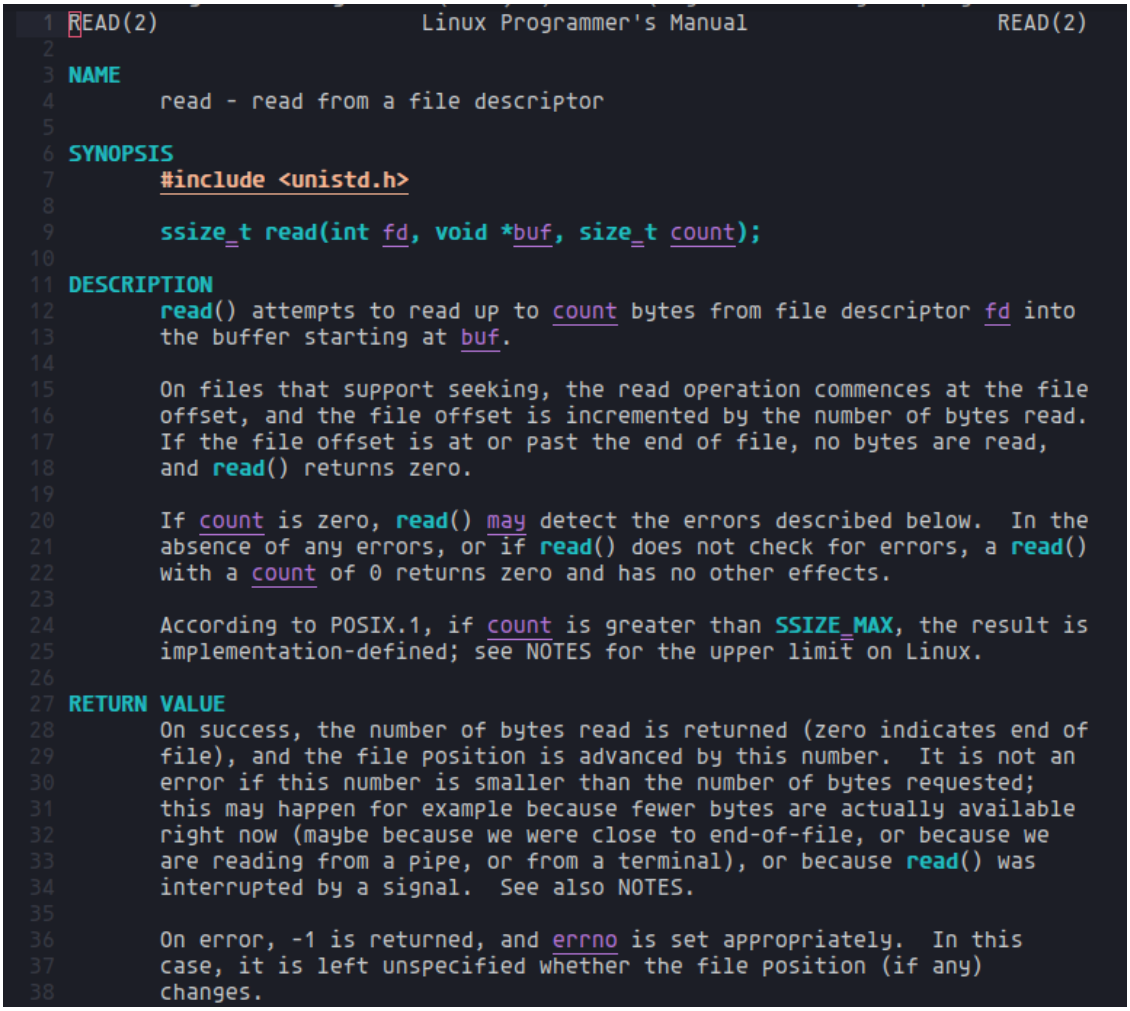
File descriptors are used by UNIX systems to abstract away most things in life. Devices are files, files are files, pipes are files, the network can be a file...

File descriptors have a very simple interface. You can read from a file descriptor, and you can write to a file descriptor.

To read from the file descriptor, you call the `read` function. Let's pull up its man page with `man 2 read` (2 because it's the second man section. Manual pages are delimited as follows:

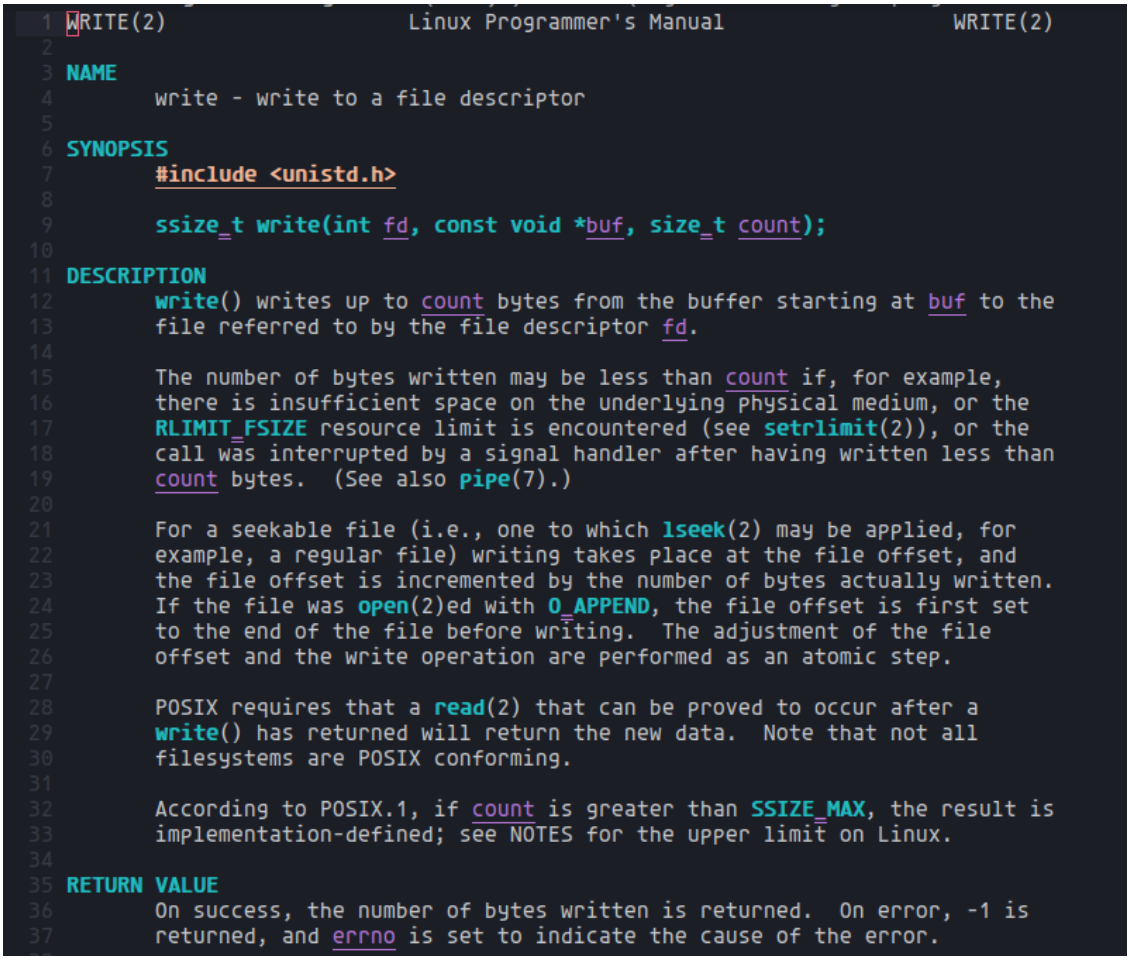
Section	Description
1	general commands
2	syscalls
3	library functions, mostly the C std library
4	special files (like devices)
5	file formats and specs
6	games and screensavers
7	misc
8	sysadmin commands and daemons

):



```
READ(2)                    Linux Programmer's Manual                    READ(2)

NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read() attempts to read up to count bytes from file descriptor fd into
       the buffer starting at buf.

       On files that support seeking, the read operation commences at the file
       offset, and the file offset is incremented by the number of bytes read.
       If the file offset is at or past the end of file, no bytes are read,
       and read() returns zero.

       If count is zero, read() may detect the errors described below.  In the
       absence of any errors, or if read() does not check for errors, a read()
       with a count of 0 returns zero and has no other effects.

       According to POSIX.1, if count is greater than SSIZE_MAX, the result is
       implementation-defined; see NOTES for the upper limit on Linux.

RETURN VALUE
       On success, the number of bytes read is returned (zero indicates end of
       file), and the file position is advanced by this number.  It is not an
       error if this number is smaller than the number of bytes requested;
       this may happen for example because fewer bytes are actually available
       right now (maybe because we were close to end-of-file, or because we
       are reading from a pipe, or from a terminal), or because read() was
       interrupted by a signal.  See also NOTES.

       On error, -1 is returned, and errno is set appropriately.  In this
       case, it is left unspecified whether the file position (if any)
       changes.
```

And, here's write's man page.



```
WRITE(2)                    Linux Programmer's Manual                    WRITE(2)

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write() writes up to count bytes from the buffer starting at buf to the
       file referred to by the file descriptor fd.

       The number of bytes written may be less than count if, for example,
       there is insufficient space on the underlying physical medium, or the
       RLIMIT_FSIZE resource limit is encountered (see setrlimit(2)), or the
       call was interrupted by a signal handler after having written less than
       count bytes.  (See also pipe(7).)

       For a seekable file (i.e., one to which lseek(2) may be applied, for
       example, a regular file) writing takes place at the file offset, and
       the file offset is incremented by the number of bytes actually written.
       If the file was open(2)ed with O_APPEND, the file offset is first set
       to the end of the file before writing.  The adjustment of the file
       offset and the write operation are performed as an atomic step.

       POSIX requires that a read(2) that can be proved to occur after a
       write() has returned will return the new data.  Note that not all
       filesystems are POSIX conforming.

       According to POSIX.1, if count is greater than SSIZE_MAX, the result is
       implementation-defined; see NOTES for the upper limit on Linux.

RETURN VALUE
       On success, the number of bytes written is returned.  On error, -1 is
       returned, and errno is set to indicate the cause of the error.
```

These manual pages tell us absolutely everything we need to know about the two commands. Isn't that nice.

When a linux process gets started, it usually has three standard file descriptors which you can use to interact with the world. These are `stdin`, `stdout`, and `stderr`. Their file descriptors are defined in `unistd.h` as `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`. `stdio.h` also defines `stdin`, `stdout`, and `stderr` as `FILE *` structures which you can pass to `stdio.h` functions. If you don't understand all this, that's fine, you can just copy what I do when writing and reading from stdin and stdout.

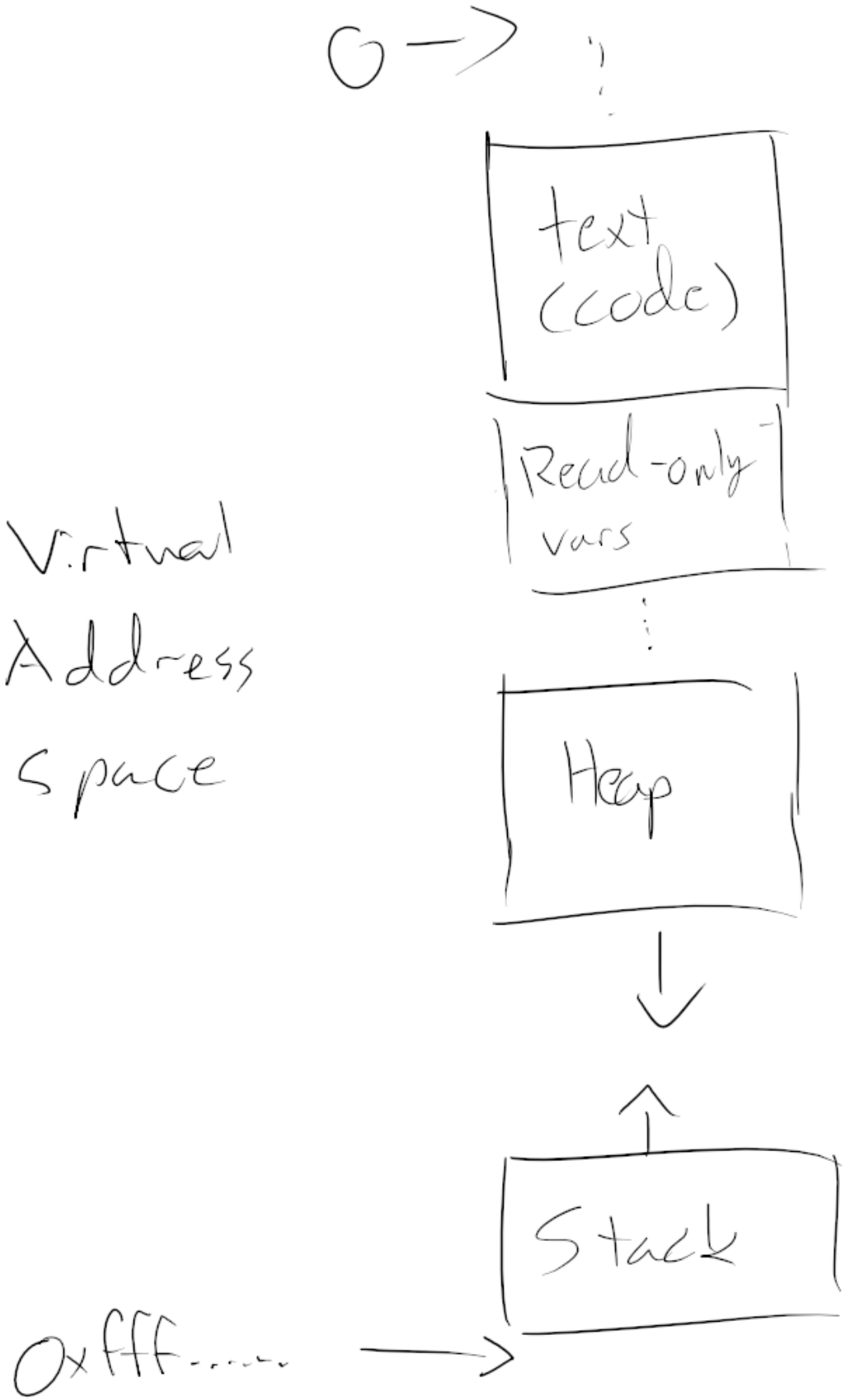
7.5.6 The stack

The stack is less important than most other things because it's mostly hidden from you as the programmer, but you have to be aware of its existence.

When a function gets called, its arguments get pushed onto the stack. What's more, its local variables are stored on the stack.

The stack starts at the end of memory (`0xffffffffffffffff`) and grows downwards, towards zero.

Here's a mental image for what it looks like:


0
text
(code)
Read-only
vars
Heap
Stack
Virtual
Address
Space
0xfff.....

A few key concepts to note:

- The stack reuses memory. If you call a function, it decreases the address of the top of the stack, and uses that location for the local variables of the function. Then, when you return from that function, it increases the address of the stack top. When you call your next function (assuming you haven't returned several times) it will use exactly the same memory for its local variables. There is no clearing that space to zero, and uninitialized local variables will have garbage in them from previous functions which have executed.
- You have to take precious care about arrays stored on the stack (local arrays of fixed size in C) because if you overrun them in a loop, you will overwrite other variables on the stack. At best, this will change your program behavior. On the average, this will crash your program when it tries to return from the function (because the return address is also on the stack). In the worst case, this is (once again) an exploit vector. Google "stack overflow" or "stack smashing." If there's no memory protection in place, this is the easiest thing to exploit.

Also, if you're looking for resources, the book "Hacking: the art of exploitation" (`https://repo.zenk-security.com/Magazine%20E-book/Hacking-%20The%20Art%20of%20Exploitation%20(2nd%20ed.%202008)%20-%20Erickson.pdf`) is probably the best intro to this kind of programming that I've ever found. It forces you to truly understand exactly how everything is implemented. Struggling through the book is hard, but the journey is well worth it.

7.5.7 `malloc/free`

This is at the bottom of the list because I didn't even use `malloc` or `free` in the program which interfaces with erlang, so you could just skip this section. So much for heap memory being required to write programs...

`malloc` is the memory allocator you can use in C. It is not the only one, as you can use `mmap`, `brk/sbrk`, `alloca`, and a few others, but 99.9% of the time, you'll be using `malloc`.

The malloc interface is simple. You pass it a number of bytes, and it'll pass you a pointer to a memory buffer that you can access of that size. Do not make any assumptions about what comes before or after it in memory. If you access those addresses, your program can crash, and you're opening yourself up to being exploited.

After you finish with a buffer in memory, you can call `free` on it. `free` takes a pointer, and will free the buffer that it is pointing to. AFAIK, you don't need to pass free back the exact pointer that is returned by `malloc`, the pointer just needs to point to some part of the buffer.

You MUST only call free ONCE. Double freeing a block of memory can crash your program, and is a common exploit vector. Google "double free exploit."

You MUST NOT access a block of memory after freeing it. Accessing a block of memory after freeing is called a "use-after-free" and is another common exploit vector. Google "use-after-free exploit."

`realloc` also exists to resize existing blocks of memory, but is slightly more complicated, so just go read the man page for it if you'd like to know more about it.

7.6 Writing the C code

To interface with the port, we simply read and write to/from stdin/stdout.

The code in the book was kinda useless, because it crashed on any input greater than 256 (because it was interpreting a general number as a byte), so I reimplemented the code in the book and the C to support variable length integers.

```c
#include <assert.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define BUF_LEN 1024
#define MAX(x, y) (((x) > (y)) ? (x) : (y))

// To enable debug messages, turn this to a 1.
#define DEBUG 1

/*
 * General notes about types:
 * - size_t is just unsigned long, and ssize_t is signed long.
 * - unsigned char -> byte.
 */

/*
 * This function reads a command from stdin. It is mostly
 * self-explanatory, and the parts which aren't, I added comments for.
 */
ssize_t read_cmd(unsigned char *buf, size_t max_len, size_t header_len) {
        ssize_t rd = read(STDIN_FILENO, buf, header_len);
        if (rd != (ssize_t) header_len) {
                fprintf(stderr, "Could not read header of cmd, rd = %ld\n", rd);
                return -1;
        }

        // if (DEBUG) fprintf(stderr, "read header of size %ld\n", rd);

        /* The following for loop turns the header_len-long header
         * into a value for our reading (i.e. it is doing the decoding
         * of a big-endian-encoded integer */
        size_t len = 0; // length of message
        for (int i = header_len - 1, exp = 0; i >= 0; --i, ++exp) {
                len |= (buf[i] << exp * 8);
        }

        // if (DEBUG) fprintf(stderr, "header says size of buf is: %lu\n", len);

        /* The following loop reads from stdin into buf until we've
         * either filled the buffer or finished reading */
        int curr_i = 0;
        while (len > 0 && (max_len - curr_i > 0)) {
                rd = read(STDIN_FILENO, buf + curr_i, MAX(max_len - curr_i, len));
                // if (DEBUG) fprintf(stderr, "read %ld bytes from stdin\n", rd);
                if (rd <= 0) { /* error or EOF */
                        fprintf(stderr,
```

```
                                "return value of %ld while reading message\n",
                                rd);
                        return -1;
                }
                curr_i += rd;
                len -= rd;
        }
        /* return how much we've read */
        return curr_i;
}

/*
 * This function outputs a hexdump to stderr.
 */
void hexdump(unsigned char *buf, size_t len) {
        int line_len = 8;
        for (size_t i = 0; i < len; ++i) {
                if (DEBUG) fprintf(stderr, "%x%x ", (buf[i] >> 4) & 0xf, buf[i] & 0xf);
                if ((i % line_len == 0) && (i != 0)) {
                        if (DEBUG) fprintf(stderr, "\n");
                }
        }
        if (DEBUG) fprintf(stderr, "\n");
}

/*
 * Decodes an int starting at the pointer provided.
 * Silently fails on len >= sizeof(unsigned long)
 */
unsigned long decode_le(unsigned char *buf, size_t len) {
        unsigned int ret = 0;
        unsigned char *ptr = buf + len - 1;
        while (ptr >= buf) { // look, pointer arithemtic
                ret <<= 8;
                ret |= *ptr--;
        }
        return ret;
}

/*
 * This function should really be trashed (because it doesn't support
 * fragmentation of result. There should be a loop very similar to the
 * read loop in read_cmd here), but it does the job technically...
 */
void write_result(unsigned long x) {
        unsigned char i = 1;
        unsigned char buf[sizeof(unsigned long) + 2];
        while (x) {
```

```
                buf[i++] = x & 0xff;
                x >>= 8;
        }
        buf[0] = i - 1;
        unsigned char head[2] = {0, i};
        if (DEBUG) {
                fprintf(stderr,
                        "C program sending the following bytes to erlang "
                        "(with header):\n");
        }
        int wr = 0;
        wr = write(STDOUT_FILENO, head, 2);
        if (wr != 2) { fprintf(stderr, "failed to write result, wr = %d\n", wr); }
        for (int j = 0; j < 2; ++j) {
                if (DEBUG) fprintf(stderr, "%x%x ", (head[j] >> 4) & 0xf, head[j] & 0xf);
        }
        wr = write(STDOUT_FILENO, buf, i);
        if (wr != i) { fprintf(stderr, "failed to write result, i = %d, wr = %d\n", i, wr); }
        for (int j = 0; j < i; ++j) {
                if (DEBUG) fprintf(stderr, "%x%x ", (buf[j] >> 4) & 0xf, buf[j] & 0xf);
        }
        if (DEBUG) fprintf(stderr, "\n");
}

int main() {
        if (DEBUG) fprintf(stderr, "starting external program\n");
        unsigned char buf[BUF_LEN];

        /* This is the event loop. Read a command, figure out which
         * command is being invoked, then send the result back. */
        ssize_t rd;
        while ((rd = read_cmd(buf, BUF_LEN, 2)) >= 0) {
                if (DEBUG) {
                        fprintf(stderr,
                                "Hexdump of bytes received by C program, minus header:\n");
                        hexdump(buf, rd);
                }

                switch (buf[0]) {
                case 1: {
                        int len_x = buf[1];
                        assert((size_t) len_x < sizeof(unsigned long));
                        int x = decode_le(buf + 2, len_x);
                        int len_y = buf[2 + len_x];
                        assert((size_t) len_y < sizeof(unsigned long));
                        int y = decode_le(buf + 3 + len_x, len_y);
                        write_result(x + y);
                } break;
```

```
                case 2: {
                        int len_x = buf[1];
                        assert((size_t) len_x < sizeof(unsigned long));
                        int x = decode_le(buf + 2, len_x);
                        write_result(x << 1);
                } break;

                default:
                        fprintf(stderr,
                                "Unrecognized function received through pipe: %d\n",
                                buf[0]);
                }
        }
}
```

7.7 Running it

Here you can see me running the code. I enabled the debug output, and you can see the bytes received and sent by the C process.

```
46> interface:start().
interface:start().
true
47> starting external program
interface:sum(11002, 1234).
interface:sum(11002, 1234).
Hexdump of bytes received by C program, minus header:
01 02 fa 2a 02 d2 04
C program sending the following bytes to erlang (with header):
00 03 02 cc 2f
12236
48> interface:sum(1234, 1).
interface:sum(1234, 1).
Hexdump of bytes received by C program, minus header:
01 02 d2 04 01 01
C program sending the following bytes to erlang (with header):
00 03 02 d3 04
1235
49> interface:twice(1234).
interface:twice(1234).
Hexdump of bytes received by C program, minus header:
02 02 d2 04
C program sending the following bytes to erlang (with header):
00 03 02 a4 09
2468
50> interface:twice(123589724).
interface:twice(123589724).
```

```
Hexdump of bytes received by C program, minus header:
02 04 5c d4 5d 07
C program sending the following bytes to erlang (with header):
00 05 04 b8 a8 bb 0e
247179448
51>
```

8 Eighth Day

Today we're working with files. There's slightly more here than meets the eye, because instead of just working with files, we're learning how to work with byte/bit streams. Erlang has great support for this, and once you get up to speed, you can write less buggy and more consise code for interacting with bytestreams. (Plus, I can imagine how this might be compiled and ran, so I haven't ran any benchmarks, but I think it might be quite fast).

8.1 Modules you gotta know

- `file` is the module that contains the normal file i/o. Opening, closing, reading, writing, ls, etc.
- `filename` is for reading and writing the names of directories and files in a platform-agnostic way.
- `filelib` is for more advanced and high-level file operations
- `io` does the actual io (although you don't need to use it to do file i/o, `file` has all you technically need). It contains routines for parsing and writing formatted data.

8.2 Reading files

There's a few ways you can read files. Here is one.

```
{lets, say, i, have}.
{some, {tuples, in}, {this, file, {and_}}, id, like, to, read, [them]}.
```

Let's say this is in a file called `asdf`, we can read the list of erlang-formatted terms within by calling `file:consult`

```
1> file:consult("./asdf").
{ok,[{lets,say,i,have},
     {some,{tuples,in},
           {this,file,{and_}},
           id,like,to,read,
           [them]}]}
```

8.2.1 More reading

If we wanted to read the terms one by one, we could use `io:read`.

```
9> {ok, S} = file:open("./asdf", read).
{ok, S} = file:open("./asdf", read).
{ok,<0.95.0>}
10> io:read(S, '').
io:read(S, '').
{ok,{lets,say,i,have}}
11> io:read(S, '').
io:read(S, '').
{ok,{some,{tuples,in},
          {this,file,{and_}},
          id,like,to,read,
          [them]}}
12> io:read(S, '').
io:read(S, '').
eof
13> file:close(S).
file:close(S).
ok
```

If we wanted to read the file line by line, we could use `io:get_line`.

```
15> {ok, S} = file:open("./asdf", read).
{ok, S} = file:open("./asdf", read).
{ok,<0.102.0>}
16> io:get_line(S, '').
io:get_line(S, '').
"{lets, say, i, have}.\n"
17> io:get_line(S, '').
io:get_line(S, '').
"{some, {tuples, in}, {this, file, {and_}}, id, like, to, read, [them]}."
18> io:get_line(S, '').
io:get_line(S, '').
eof
```

8.2.2 Reading with pread

You can also use pread to do random access in a file. After opening a file with `raw`, as shown, you can use `file:pread` to read a specified number of bytes starting at a specified offset.

```
19> {ok, F} = file:open("./asdf", [read, binary, raw]).
{ok, F} = file:open("./asdf", [read, binary, raw]).
{ok,{file_descriptor,prim_file,
                     #{handle => #Ref<0.3399782956.215089163.90801>,
                       owner => <0.81.0>,r_ahead_size => 0,
                       r_buffer => #Ref<0.3399782956.215089154.90797>}}}
20> file:pread(F, 10, 10).
file:pread(F, 10, 10).
{ok,<<", i, have}">>}
```

```
21> file:pread(F, 20, 10).
file:pread(F, 20, 10).
{ok,<<".\n{some, {">>}
22> file:pread(F, 10, 20).
file:pread(F, 10, 20).
{ok,<<", i, have}.\n{some, {">>}
23> file:pread(F, 15, 50).
file:pread(F, 15, 50).
{ok,<<"have}.\n{some, {tuples, in}, {this, file, {and_}}, ">>}
24> file:pread(F, 15, 100).
file:pread(F, 15, 100).
{ok,<<"have}.\n{some, {tuples, in}, {this, file, {and_}}, id, like, to, read, [them]}.">>}
25> file:pread(F, 15, 200).
file:pread(F, 15, 200).
{ok,<<"have}.\n{some, {tuples, in}, {this, file, {and_}}, id, like, to, read, [them]}.">>}
2
```

8.2.3 Reading and writing entire files with binaries

Erlang has a really cool built-in data structure called a Binary.

you can use `file:read_file` to read an entire file into a binary string atomically. This is by far the most efficient way, and so can be used to great effect.

This is how I do the file i/o in the bencode parser that I wrote earlier which you can find in `./code/torrent/benc.erl`.

8.3 Writing files

The most commonly used bif to create formatted output is `io:format`. `io:format` is the 'common' printf just like in other languages, but with slightly different format string syntax.

You can call it as

```
-spec io:format(IODevice, Format, Args) -> ok.
```

`IODevice` in this case is a device which you have opened in write mode. You can also product strings by calling it without an `IODevice`.

Here are some format options you can use:

- `~n` Write a line feed. This does the right thing on various platforms (CLRF on windows)
- `~p` Pretty-print the argument
- `~s` When the arg is either a string or IO List (which is a list of strings or bytes or binaries afaik), this will print it without any quotation marks.
- `~w` Write data with the standard syntax. You use this to output erlang terms.

There's more here than meets the eye, as these can also take extra arguments, but we can't cover them all here:

```
Format                          Result
======                          ======
```

```
io:format("|~10s|",["abc"])         |       abc|
io:format("|~-10s|",["abc"])        |abc       |
io:format("|~10.3.+s|",["abc"])     |+++++++abc|
io:format("|~-10.10.+s|",["abc"])   |abc+++++++|
io:format("|~10.7.+s|",["abc"])     |+++abc++++|
```

These are copy pasted from the book. Pls don't sue me.

8.4 `file:write_file`

`file:write_file(File, IO)` deserves its own section, because it is fast, atomic, and easy. `File` is a file, and `IO` is an I/O list, which is a list of binaries, integers from 0 to 255 (bytes), or other I/O lists (this includes strings). When you pass a deeply nested list, it is flattened without you having to do anything about it. This is very helpful as shown again in the bencode parser in `./code/torrent/benc.erl`.

8.5 Directory operations

To operate on directories, erlang gives us `list_dir`, `make_dir` and `del_dir`. They all do what you'd expect.

8.5.1 File info

To find info about a file, we call `file:read_file_info` which returns `{ok, Info}` on success. Info then returns a `#file_info` struct.

To access this, I think you need to include the `"kernel/include/file.hrl"` to import the `#file_info` record.

Otherwise, it works exactly as you'd expect.

Go read the docs or source if you want to know what information is exposed here. It's not terribly exciting, so I won't discuss it here, but everything you'd need is here.

8.5.2 Copy and delete

`file:copy(Source, Dest)` copies Source to Dest and `file:delete(Dest)` deletes Dest.

8.6 Let's clone `find` as an exercise.