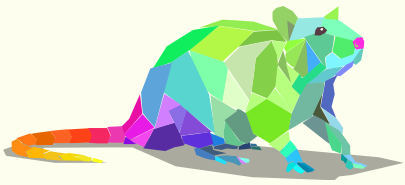


Forking is Cool, or: A Unix Shell in Zig

Created: 2021-06-02

Or: "Dave Learns How to Fork"

In this article, I make a real, working interactive shell written in Zig. Though essentially a toy, it demonstrates the incredible elegance of the `fork()` system call and is an example of what can be done with **absolutely no runtime memory allocations**.

Note	You can also go straight to the repo to view the source of zigish. But please keep in mind that it's a toy shell. Its value is in the learning exercise.

A bucket list program

The day has apparently come. With trembling hands, I reached into my trunk of "programs to write when I finally learn a systems language" and picked a bookmark from the top of the heap: Write a Shell in C. A shell! Awesome. This is one of my "bucket list" programs - like a game, OS, text editor, or programming language (but much easier than any of those).

`main()`

I started by following along with the C tutorial. A `shellLoop()` function does the real work:

```
const std = @import("std");

pub fn main() !u8 {
    const stdin = std.io.getStdIn().reader();
    const stdout = std.io.getStdOut().writer();
    try stdout.print("*** Hello, I am a real shell! ***\n", .{});
    try shellLoop(stdin, stdout);

    return 0;
}
```

My `main()` has a little more in it because, unlike C, the Zig standard library does not have implicit handles to STDIN and STDOUT. So I get a Reader and Writer to those handles and pass them to my loop function. (In fact, there's a *lot* of stuff that is implicit in C and explicit in Zig. I like this "nothing hidden" approach.)

The `shellLoop()` contains exactly what you think it does: an infinite loop:

```
fn shellLoop(stdin: std.fs.File.Reader, stdout: std.fs.File.Writer) !void {
    while (true) {
        ...
    }
}
```

If you're new to Zig, the `!u8` and `!void` return types of these two functions are examples of error union types. That is, the return value will *either* be the specified value *or* an error. You can also specify what type of error can be returned: `FooError!ReticulatedFoo`.

Even though it's short (at the moment, the whole thing is just 45 lines! `sed '/^$/d; /^ *\/\//d' src/main.zig | wc -l`), I'm not going to go through every single line of the program.

You can view the full source of zigish here.

This is just the cool parts.

Prompt

Let's go with a minimalist shell prompt because this is a minimalist shell:

```
try stdout.print("> ", .{});
```

Again, if you're new to Zig, `try` will pass any errors from `print()` up the call stack (to `main()`, in this case).This is why we must have the `!void` error return union type.

The `.{}` syntax in the second parameter is a "tuple" (an anonymous struct) which would hold any values we wished to print in the first parameter, the format string.

Read the user input

Let's get the user's input from the stdin Reader:

```
var input_buffer: [max_input]u8 = undefined;

var input_str = (try stdin.readUntilDelimiterOrEof(input_buffer[0..], '\n')) orelse {
    // No input, probably CTRL-d (EOF). Print a newline and exit!
    try stdout.print("\n", .{});
    return;
};
```

Here, the `readUntilDelimiterOrEof()` function takes a slice (pointer with length) to a chunk of memory I've statically allocated in the array `input_buffer`. It reads input (in "cooked mode") until it hits the newline delimiter character. Then it returns a new slice containing the input.

Next, we need to split the input into a command and its arguments. I'll do that in the simplest possible way using the `split()` function which returns a `SplitIterator`:

```
var tokens = std.mem.split(input_str, " ");

while (tokens.next()) |tok| {
    ...
}
```

We'll come back to what, exactly, is in this loop in a moment. For now, just know that we have split up the input string and stored the pieces: `ls -al foo` becomes `ls`, `-al`, and `foo`.

Fork!

We have the command and any arguments. We're ready to fork a new process and execute the command.

This is the really cool part.

```
const fork_pid = try std.os.fork();

if (fork_pid == 0) {

    // We are the child, execute the command.

} else {

    // We are the parent, wait for the child to exit.
}
```

I don't know about you, but I stared at that `if` statement for a while. How could the process ID (PID) be two different things in the same program?

From the Linux `fork(2)` man page:

```
fork() creates a new process by duplicating the calling process.
The new process is referred to as the child process.  The calling
process is referred to as the parent process.

The child process and the parent process run in separate memory
spaces.  At the time of fork() both memory spaces have the same
content.
```

So after we call `fork()` above, we now have two nearly identical processes running the same code and with the same values in memory.

The two processes also *continue running* at the exact same point: returning from `fork()`.

- One of them is the parent process, for which the return value is the PID of the child process.
- The other one is the child process, for which the return value is 0.

The `if` statement is how we tell the parent process to do one thing and the child process to do another.

I think the elegance and simplicity of this approach is just absolutely beautiful.

Note	You might be wondering if duplicating the process and all of its running state is terribly inefficient. I wondered that too. Modern implementations of `fork()` use copy-on-write to only duplicate the data if it is actually modified. Until then, it is actually the *same* data in memory. As for the program's instruction code, it is in a read-only segment, so it never needs to be duplicated.

The parent waits for the child

For this simple shell, the parent's job is easy. We simply wait for the child to exit.

The `waitpid()` system call waits for a specific process (by PID) to have a state change (terminated, stopped, resumed). Here it is in our post-fork `if` branch for the parent:

```
if (fork_pid == 0) {

    // We are the child, execute the command.

} else {
    const wait_result = std.os.waitpid(fork_pid, 0);
    if (wait_result.status != 0) {
        try stdout.print("Command returned {}.\n", .{wait_result.status});
    }
}
```

If the child's return status is anything but `0`, that indicates an error, so we'll print it out.

The child replaces itself with the command

The duplicate child process now has an interesting job: replacing itself with the requested command. We do this with the `exec()` family of system calls.

Now we can fill in the child portion of the post-fork `if` branch:

```
if (fork_pid == 0) {
    const result = std.os.execvpeZ(args_ptrs[0].?, &args_ptrs, &env);
    try stdout.print("ERROR: {}\n", .{result});
} else {
    const wait_result = std.os.waitpid(fork_pid, 0);
    if (wait_result.status != 0) {
        try stdout.print("Command returned {}.\n", .{wait_result.status});
    }
}
```

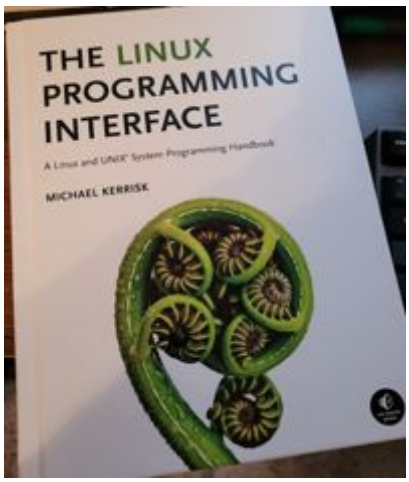
This uses the `execvpeZ()` variant from the Zig standard library which supplies arguments and any environment data to the new process. This variant also uses `$PATH` to resolve commands without a `/` in the name.

So `exec()` calls aren't quite as mind-blowing as `fork()`, but they're still quite interesting to think about. After a successful call to an `exec()` function, none of the rest of the code in the child process will ever run. It has been replaced by the new command process!

That's why we print an `ERROR:` message on the next line. The only reason that line would ever execute is because the `exec()` call has failed.

The parent continues the loop

Once the child exits, the `waitpid()` call returns and the parent continues running, looping back to the prompt and getting the next command from the user.



By the way, I got the basic structure of this shell from the C tutorial, but for a full understanding of the system calls, I read Chapters 24-27 in *The Linux Programming Interface* by Michael Kerrisk. I look forward to spending more quality time with this excellent book.

How to `[*:null]const ?[*:0]const u8`

I mentioned that I would get back to the innards of that `SplitIterator` loop where we do something with the command and argument chunks of the user input string.

I don't mind saying, it took me one whole evening, a night of sleep, and then a morning to get that part right. The reason is that I became obsessed with the idea that my shell would not make any dynamic memory allocations at runtime. And that's why I chose `execvpeZ()` from the Zig Standard Library to interface with the `exec()` call. It doesn't require an allocator. To avoid allocators, it requires that you supply it with sentinel-terminated many-item pointers.

The principle is simple enough: a many-item pointer is Zig's way of saying, "this points to some unknown number of items of this type (with a known size)." To use a many-item pointer safely, you need to either know the exact size, *OR* you put a sentinel value at the end which acts as a stop sign when you reach it.

The syntax for a many-item pointer to a list of `u8` values is `[*]u8`.

The syntax for a many-item pointer to a list of `u8` values terminated by sentinel value `0` is `[*:0]`.

(Zig lets you choose any value of the child type as the sentinel value. You could use `81` or `'Q'` and the letter 'Q' would be the sentinel value!)

Well, for the command argument parameter `argv_ptr`, `execvpeZ()` wants a value of type `[*:null]const ?[*:0]const u8` which is a "null-terminated many-item pointer to a list of constant optional zero-terminated many-item pointers to lists of constant unsigned eight-bit integers.

Let's break that down into byte-sized pieces:

- `[*:null]` means we're pointing to a list of items with a `null` terminator.
- `const` means the values will not be changed inside `execvpeZ()`.
- `?` means each of the following values are "optional", which is to say they may be `null`. This is required because the sentinel value for the many-item pointer must be a value of the child type!
- `[*:0]` means each of the (optional) values are going *themselves* be zero-terminated many-item pointers (zero is the "NULL character" in ASCII and Unicode).
- `const` means none of the pointed-to items will be changed inside `execvpeZ()`.
- `u8` means the actual data we're pointing to is a sequence of unsigned eight-bit integers (which is how we store ASCII or UTF-8 encoded Unicode values).

No problemo, right?

We can skip my failed attempts and go straight to what I ended up with:

```
var args: [max_args][max_arg_size:0]u8 = undefined;
var args_ptrs: [max_args:null]?[*:0]u8 = undefined;

...

var i: usize = 0;
while (tokens.next()) |tok| {
    std.mem.copy(u8, &args[i], tok);
    args[i][tok.len] = 0; // add sentinel 0
    args_ptrs[i] = &args[i];
    i += 1;
```

```
}
args_ptrs[i] = null; // add sentinel null
```

I use the `std.mem.copy()` function to copy the bytes from the slice returned by the `SplitIterator` `next()` method to the `args` storage array.

I manually add a `0` value after the copied bytes. Zig only enforces that the array *ends* with a terminal, but it does not automatically insert them inside the array for us because it has no way of knowing this is what we want (for all it knows, we want to append another string after this one).

Then I immediately store a pointer to the storage array slot in my `args_ptrs` array.

(In some of my earlier failed attempts, I tried to go straight from `[N][M:0]u8` to `[*:null][*:0]u8` which proved to be a lot of mental gymnastics, a surprising amount of code, and I never quite got it right.)

Let's take another look at where I invoke the `exec()` with this info:

```
const result = std.os.execvpeZ(args_ptrs[0].?, &args_ptrs, &env);
```

The first parameter (the command) is the first item from my `args_ptrs` array. I "unwrap" the optional (possibly null) pointer with `.?`. If it *is* null, the program will crash.

The second parameter is just the address of the `args_ptrs` array itself. (It is expected, but not required, that it be identical to the first parameter).

Finally, the last parameter is the address of another structure containing the environment (you know, environment variables like `$PATH`). For this toy, I'm just passing in null. But not just any null, a null of this type:

```
const env = [_:null]?[*:0]u8{null};
```

And that's not just bookkeeping. It matters how big the null is (`0000` and `00000000` take up different amounts of memory).

To see an example of this problem in action, check out this article I came across (during one of my failed attempts to get my args pointer right that resulted in an `EFAULT` error): the exec that failed to exec.

Playing with my new toy shell

Let's play with `zigish`! By the way, the name stands for "Zig Interactive Shell", but I'm also playing on the "ish" modifier in English meaning "mostly, kinda, sorta".

```
$ zig build
$ zig-cache/bin/zigish
*** Hello, I am a real shell! ***
> ls
README.adoc  build.zig  src  zig-cache
> date
Sat Jun  5 13:26:59 EDT 2021
> foo
ERROR: error.FileNotFound
>
>
> ^D
$
```

And we can run Bash *in* Zigish:

```
$ zig-cache/bin/zigish
*** Hello, I am a real shell! ***
> bash
$ pstree dave

bash---zigish---bash---pstree

$ ^D
> ^D
$
```

It's also worth noting that anything which requires a particular environment may have problems. I got this error when attempting to run `zig build` from zigish: `AppDataDirUnavailable`. I traced that one down to the fact that I'm not passing in the `$PATH` environment variable to the child process when I `exec()`. Eh, no big deal.

Lastly (and for some reason, this has always been part of what makes this a "bucket list" program), I wanted to set this as the login shell for a user.

First I'll make a symlink to it in `/usr/bin/` to give it that official flair:

```
$ sudo ln -s /home/dave/proj/zig/zigish/zig-cache/bin/zigish /usr/bin/
```

And make a new user:

```
$ sudo useradd --shell /usr/bin/zigish ziggy
$ sudo passwd ziggy
$ su ziggy
*** Hello, I am a real shell! ***
>
```

And yes, I really did reboot and log in as the user 'ziggy' afterward just to see it "for real".

Conclusion

I learned a lot making this and it was (mostly) a ton of fun. I highly recommend writing your own shell at some point. It's one of the biggest bang-for-your-buck projects you could hope to do.

Definitely give Zig a shot if you're looking for a C replacement. It's still a work in progress and things are rapidly changing, but it has one of the best communities I've ever seen.

Okay, so, shell done! Now for the text editor, operating system, game(s), programming language(s), and...uh a Web browser. :-)

[<< My Zig-related stuff.]