Extracted from:

Modern Systems Programming with Scala Native

Write Lean, High-Performance Code without the JVM

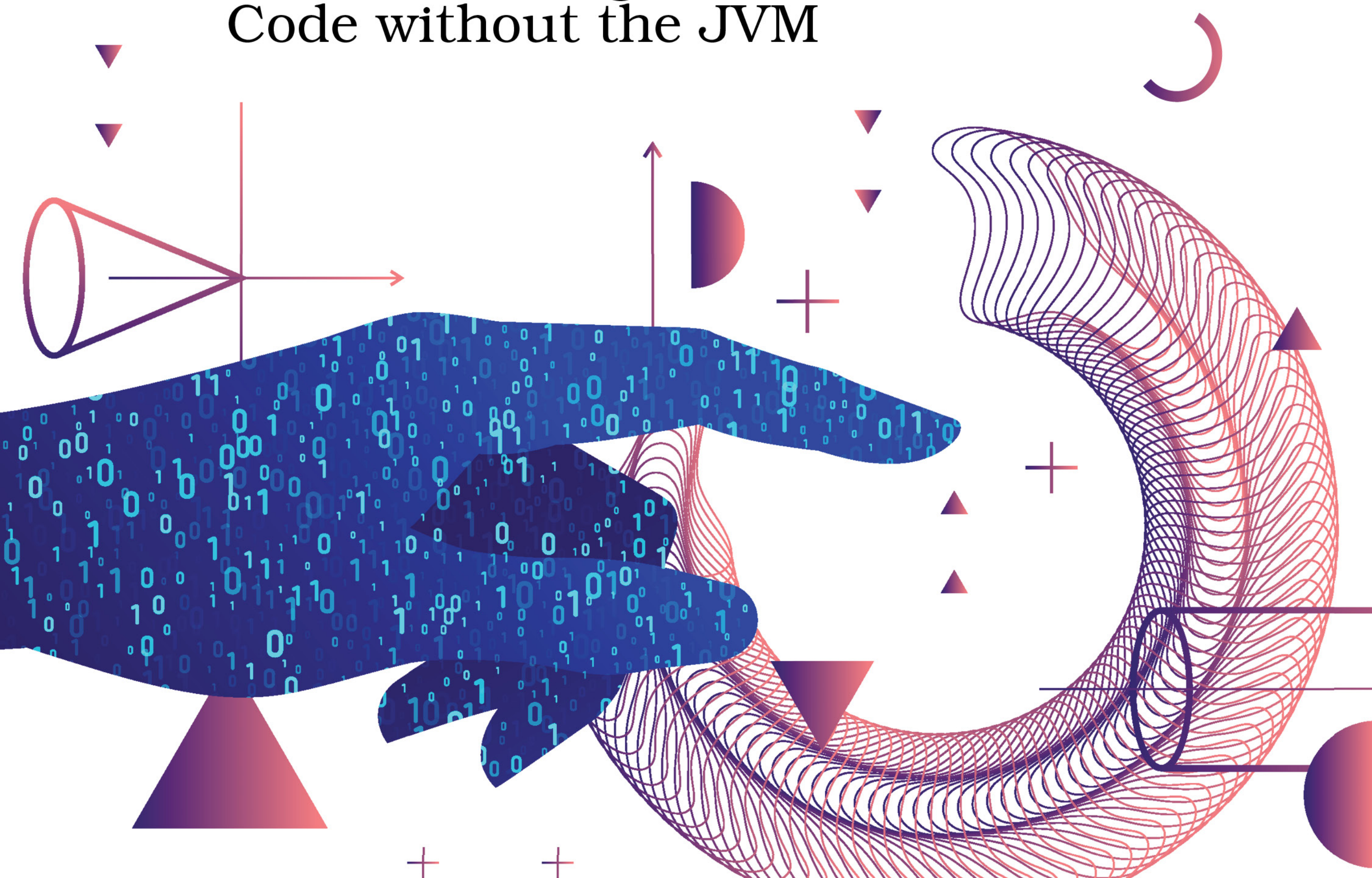
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Modern Systems Programming with Scala Native

Write Lean, High-Performance Code without the JVM



Richard Whaling

Foreword by Denys Shabalin, Author of Scala Native

Edited by Katharine Dvorak

Modern Systems Programming with Scala Native

Write Lean, High-Performance Code without the JVM

Richard Whaling

The Pragmatic Bookshelf
Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Development Editor: Katharine Dvorak
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-622-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2020

Introducing Concurrency with fork() and wait()

In a UNIX-like OS, processes are traditionally created with the system call `fork()`. `fork()` doesn't create a process out of thin air; instead, it creates a copy of the process that calls it, which will have access to all of the calling process's state and code. This is for a good reason: even if we're going to call `exec`, we need some way to control the behavior of the new process before `exec` is called. `fork()` allows us to both *create* a new process and *coordinate* its behavior with the rest of our code.

Its signature is simple:

```
def fork():Int
```

`fork()` takes no arguments and returns an `Int`. Unlike every other function we've discussed, and probably unlike every function you've ever written, `fork()` *returns twice*. It returns exactly once in the calling process, and it returns exactly once in the newly created process.

fork vs clone

> Although fork is a low-level concurrency primitive, `fork()` itself is, surprisingly, not a system call. Just like `malloc()` wraps the system call `sbrk()`, `fork()` likewise wraps a system call named `clone()` that is similarly unsuited to use by humans. `clone()` is responsible for creating new processes, as well as new threads, and can control the isolation of units of execution in a more fine-grained fashion than we'll need to.

What is particularly unusual is that it returns different values to the two processes. In the calling process, it returns the *process id* of the newly created process—an integer that uniquely identifies the process for as long as it exists in the system's process table. In the new process, `fork()` instead returns `0`. By inspecting the return value of `fork()`, we can thus determine which of the two new processes we are in.

To wrap `fork()` in a way that is suitable to idiomatic Scala, let's just pass it a runnable task, and then return the resulting PID in the parent while ensuring that the child terminates after completing its task:

ForkWaitShell/nativeFork/nativeFork.scala

```scala
def doFork(task:Function0[Int]):Int = {
  val pid = fork()
  if (pid > 0) {
    pid
  } else {
```

```
    val res = task.apply()
    stdlib.exit(res)
    res
  }
}
```

Note, however, that when we execute `doFork()`, the parent will return immediately, while the child is still running, which means we'll need to be very careful about how we proceed. All modern operating systems take responsibility for deciding *when* processes run, *where* they run, and for how long. We saw this in Chapter 3, Writing a Simple HTTP Client, on page ?, when we observed that other programs would run while ours was blocked waiting for I/O. And in a multicore operating system, not only will both processes proceed with their programs separately, in any order, they may also execute at the same time. This is called *preemptive multitasking*, and it can require a certain amount of defensive coding. For example, could a "race condition" emerge with unintended behaviors if your two processes are executed in a different order than you expected? Fortunately, we have powerful tools to coordinate the work of our processes.

First, there's `getpid()` and `getppid()`:

```
def getpid():Int
def getppid():Int
```

`getpid()` simply returns the process id of the process that calls it. This will be useful for understanding the behavior of complex chains of processes.

`getppid()` returns the pid of the *parent process* when it's called. Because processes are created by `fork`, every process should have a parent process. In some cases, however, a parent may exit before a child, in which case either the "orphaned" child process may be terminated, or else it will be "adopted" by PID 1, the `init` process.

Process Groups and Sessions

In addition to a parent process, UNIX processes also belong to process groups and sessions. Typically, these are used for scenarios such as ensuring that all processes spawned by a terminal session terminate at the same time as the original terminal. This book won't deal with process groups or sessions in depth, but you can refer to the manual for your favorite UNIX OS for more details.

Finally, we must consider `wait()` and `waitpid()`:

```
def wait(status:Ptr[Int]):Int
def waitpid(pid:Int, status:Ptr[Int], options:Int):Int
```

```
def check_status(status:Ptr[Int]):Int
```

`wait()` is the essential function for synchronizing processes. When called, it blocks until a child of the calling process completes, sets its return code in `status`, and returns the `pid` of the completed child process. `waitpid` simply provides more options: the argument `pid` can take either the `pid` of a specific child process, `0` to wait for any child in the same process group, `-1` to wait for any child group at all, and `-k` to wait for any child in process group `k`. Likewise, `options` can take several flags, most important of which is `WNOHANG`, which prevents `waitpid()` from blocking, and instead returns `0` immediately if no children are exited.

One quirk in the case of certain anomalous exit conditions is that the `status` return may have multiple values, packed bit-wise into a 4-byte integer address. Although these can be unpacked manually, it's usually best to rely on your OS's facilities for doing so. In Scala Native, these are packaged by the `check_status` function, which will return the exit code of a terminated process, given a `status` value. For our purposes, it's sufficient to just check that `status` is nonzero.

Waiting Is Mandatory

If you're creating processes with `fork()`, it's essential that you plan to call `wait()` for each one. Completed child processes keep their exit code in the kernel's process table until `wait()` is called. These so-called zombie processes can overwhelm and crash a system, even outside of container boundaries, if they're allowed to grow unchecked.

And if we put these together with some boilerplate code to check the different reasons for termination, we get the following:

ForkWaitShell/nativeFork/nativeFork.scala

```
def await(pid:Int):Int = {
  val status = stackalloc[Int]
  waitpid(pid, status, 0)
  val statusCode = !status
  if (statusCode != 0) {
    throw new Exception(s"Child process returned error $statusCode")
  }
  !status
}
```

Now we have the basic ingredients in place to launch and monitor commands, just like a shell! All we have to do is stitch `runCommand`, `doFork`, and `await` together, and then it's straightforward to use if we can pass in some string arguments:

ForkWaitShell/nativeFork/nativeFork.scala

```
def doAndAwait(task:Function0[Int]):Int = {
  val pid = doFork(task)
  await(pid)
}
```

ForkWaitShell/nativeFork/nativeFork.scala

```
def main(args:Array[String]):Unit = {
  if (args.size == 0) {
    println("bye")
    stdlib.exit(1)
  }

  println("about to fork")

  val status = doAndAwait { () =>
    println(s"in child, about to exec command: ${args.toSeq}")
    runCommand(args)
  }
  println(s"wait status ${status}")
}
```

When run, we get the following output:

```
$ ./target/scala-2.11/nativefork-out /bin/ls -l
about to fork
in child, about to exec command: WrappedArray(/bin/ls, -l)
build.sbt               nativeFork.scala        project         target
wait status 0
```

Success! Now we can execute programs, just like a shell. However, a shell can do more than run single programs; some of the most powerful shell capabilities involve running multiple programs in different configurations and routing their inputs and outputs in a controlled fashion. So, how do we implement these patterns in Scala Native?