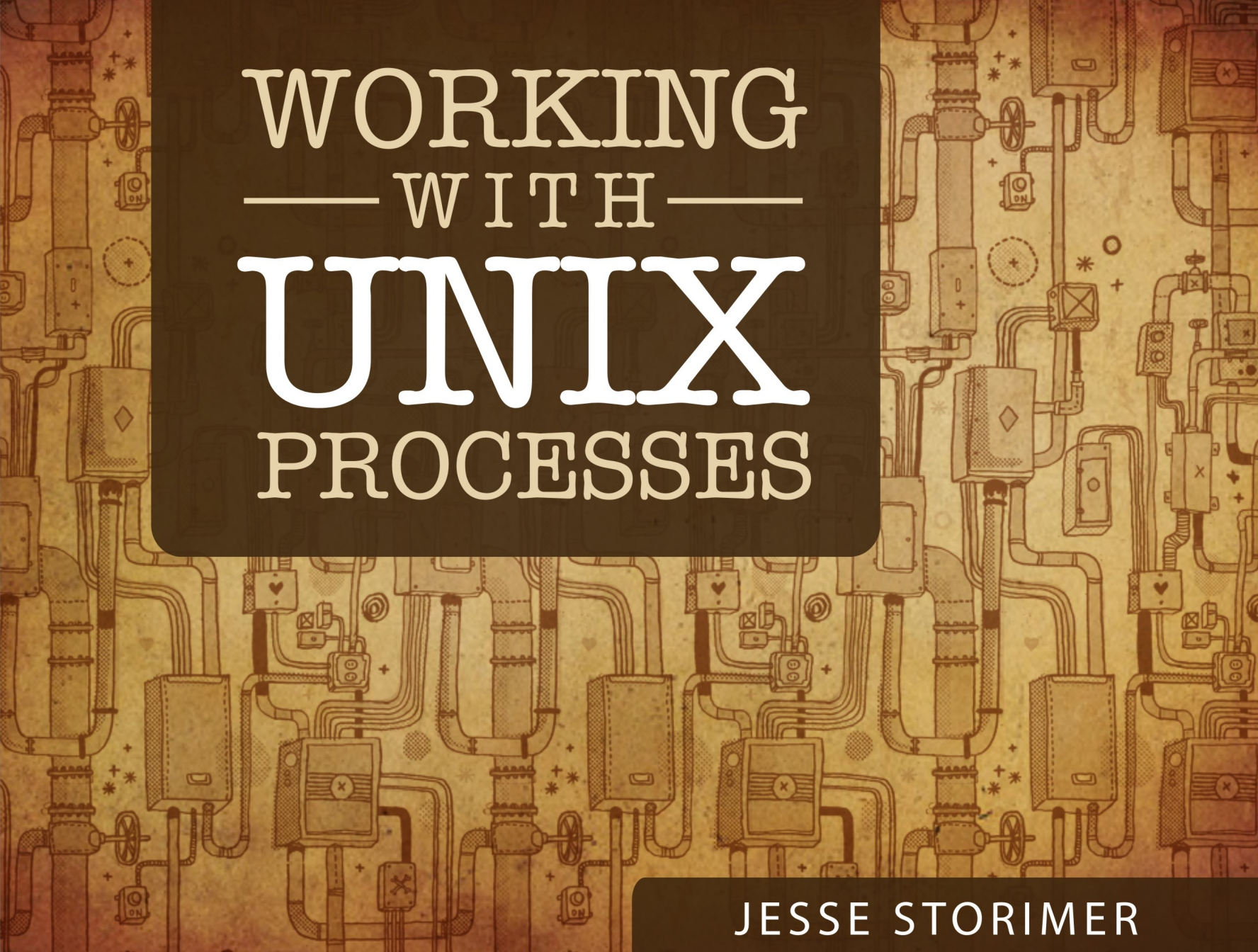
WORKING
WITH
UNIX
PROCESSES
JESSE STORIMER

Working with Unix Processes

This is a one-man operation, please respect the time and effort that went into this book. If you came by a free copy and find it useful, you can compensate me at http://workingwithunixprocesses.com.

Acknowledgements

A big thank you to a few awesome folks who read early drafts of the book, helped me understand how to market this thing, gave me a push when I needed it, and were all-around extremely helpful: Sam Storry, Jesse Kaunisviita, and Marc-André Cournoyer.

I have to express my immense gratitude towards my wife and daughter for not only supporting the erratic schedule that made this book possible, but also always being there to provide a second opinion. Without your love and support I couldn't have done this. You make it all worthwhile.

Contents

Updates

- December 20, 2011 - First public version
- December 21, 2011 - Typos
- December 23, 2011 - Explanation for Process.setsid
- December 27, 2011 - Section on SIGCHLD and concurrency
- December 27, 2011 - Note about redefining 'default' signal handlers
- December 28, 2011 - Typos
- December 31, 2011 - Clarification around exiting with Kernel.raise; Section on using fork with a block; More typos; Note about getsid(2)
- January 13, 2012 - Improved code highlighting. Improved e-reader formatting.
- February 1, 2012 - New cover art.
- February 7, 2012 - New chapters: zombie processes, environment variables, preforking servers, the spyglass project. Clarifications on CoW-friendliness in MRI. Added sections for IO.popen and Open3.
- February 13, 2012 - Clarifications on Process::WNOHANG and file descriptor relations.
- March 13, 2012 - Include TXT format.
- March 29, 2012 - Formatting and errata.
- April 20, 2012 - New chapters: ARGV and IPC.
- May 15, 2012 - Clarification about reentrancy in signal handlers.

- June 12, 2012 - New chapter on rlimits; Many formatting/syntax updates.
- Dec 4, 2012 - Fixed ToC target page issue; Fixed lsof reference.
- July 30, 2013 - Updated CoW section to reflect MRI's new GC; Added bit about FD leaking to fork + exec section.

Chapter 1

Introduction

When I was growing up I was sitting in front of a computer every chance I got. Not because I was programming, but because I was fascinated by what was possible with this amazing machine. I grew up as a computer user using ICQ, Winamp, and Napster.

As I got older I spent more time playing video games on the computer. At first I was into first-person shooters and eventually spent most of my time playing real-time strategy games. And then I discovered that you can play these games online! Throughout my youth I was a 'computer guy': **I knew how to use computers, but I had no idea how they worked under the hood.**

The reason I'm giving you my background is because I want you to know that I was not a child prodigy. I did not teach myself how to program Basic at age 7. When I took my first computer programming class I was not teaching the teacher and correcting his mistakes.

It wasn't until my second year of a University degree that I really came to love programming as an activity. Some may say that I'm a late bloomer, but I have a feeling that I'm closer to the norm than you may think.

Although I came to love programming for the sake of programming itself I still didn't have a good grasp of how the computer was working under the hood. If you had told me back then that all of my code ran inside of a *process* I would have looked at you sideways.

Fortunately for me I was given a great work opportunity at a local web startup. This gave me a chance to do some programming on a real production system. This changed everything for me. This gave me a reason to learn how things were working under the hood.

As I worked on this high-traffic production system I was presented with increasingly complex problems. As our traffic and resource demands increased **we had to begin looking at our full stack to debug and fix outstanding issues**. By just focusing on the application code we couldn't get the full picture of how the app was functioning.

We had many layers in front of the application: a firewall, load balancer, reverse proxy, and http cache. We had layers that worked alongside the application: job queue, database server, and stats collector. Every application will have a different set of components that comprise it, and this book won't teach you everything there is to know about all of it.

This book will teach you all you need to know about Unix processes, and that is **guaranteed to improve your understanding of any component at work in your application**.

Through debugging issues I was forced to dig deep into Ruby projects that made use of Unix programming concepts. Projects like Resque and Unicorn. These two projects were my introduction to Unix programming in Ruby.

After getting a deeper understanding of how they were working I was able to diagnose issues faster and with greater understanding, as well as debug pesky problems that didn't make sense when looking at the application code by itself.

I even started coming up with new, faster, more efficient solutions to the problems I was solving that used the techniques I was learning from these projects. Alright, enough about me. Let's go down the rabbit hole.

Chapter 2

Primer

This section will provide background on some key concepts used in the book. It's definitely recommended that you read this before moving on to the meatier chapters.

Why Care?

The Unix programming model has existed, in some form, since 1970. It was then that Unix was famously invented at Bell Labs, along with the C programming language or framework. In the decades that have elapsed since then Unix has stood the test of time as the operating system of choice for reliability, security, and stability.

Unix programming concepts and techniques are not a fad, they're not the latest popular programming language. These techniques transcend programming languages. Whether you're programming in C, C++, Ruby, Python, JavaScript, Haskell, or [insert your favourite language here] these techniques WILL be useful.

This stuff has existed, largely unchanged, for decades. Smart programmers have been using Unix programming to solve tough problems with a multitude of programming languages for the last 40 years, and they will continue to do so for the next 40 years.

Harness the Power!

I'll warn you now, the concepts and techniques described in this book can bring you great power. With this power you can create new software, understand complex software that is already out there, even use this knowledge to advance your career to the next level.

Just remember, with great power comes great responsibility. Read on and I'll tell you everything you need to know to gain the power and avoid the pitfalls.

Overview

This book is not meant to be read as a reference manual. It's more of a walkthrough. To get the most out of it you should read it sequentially, since each chapter builds on the last. Once you're finished you can use the chapter headings to find information if you need a refresher.

This book contains many code examples. I highly recommend that you follow along with them by actually running them yourself in a Ruby interpreter. Playing with the code yourself and making tweaks will help the concepts sink in that much more.

Once you've read through the book and played with the examples I'm sure you'll be wanting to get your hands on a real world project that's a little more in depth. At that point have a look at the included Spyglass project.

Spyglass is a web server that was created specifically for inclusion with this book. It's designed to teach Unix programming concepts. It takes the concepts you learn here

and shows how a real-world project would put them to use. Have a look at the last chapter in this book for a deeper introduction.

System Calls

To understand system calls first requires a quick explanation of the components of a Unix system, specifically userland vs. the kernel.

The kernel of your Unix system sits atop the hardware of your computer. It's a middleman for any interactions that need to happen with the hardware. This includes things like writing/reading from the filesystem, sending data over the network, allocating memory, or playing audio over the speakers. Given its power, programs are not allowed direct access to the kernel. Any communication is done via system calls.

The system call interface connects the kernel to userland. It defines the interactions that are allowed between your program and the computer hardware.

Userland is where all of your programs run. You can do a lot in your userland programs without ever making use of a system call: do mathematics, string operations, control flow with logical statements. But I'd go as far as saying that if you want your programs to do anything interesting then you'll need to involve the kernel via system calls.

If you were a C programmer this stuff would probably be second nature to you. System calls are at the heart of C programming.

But I'm going to expect that you, like me, don't have any C programming experience. You learned to program in a high level language. When you learned to write data to the filesystem you weren't told which system calls make that happen.

The takeaway here is that system calls allow your user-space programs to interact indirectly with the hardware of your computer, via the kernel. We'll be looking at common system calls as we go through the chapters.

Nomenclature, wtf(2)

One of the roadblocks to learning about Unix programming is where to find the proper documentation. Want to hear the kicker? It's all available via Unix manual pages (manpages), and if you're using a Unix based computer right now it's already on your computer!

If you've never used manpages before you can start by invoking the command `man man` from a terminal.

Perfect, right? Well, kind of. The manpages for the system call api are a great resource in two situations:

1. you're a C programmer who wants to know how to invoke a given system call, or
2. you're trying to figure out the purpose of a given system call

I'm going to assume we're not C programmers here, so #1 isn't so useful, but #2 is very useful.

You'll see references throughout this text to things like this: select(2). This bit of text is telling you where you can find the manpage for a given system call. You may or may not know this, but there are many sections to the Unix manpages.

Here's a look at the most commonly used sections of the manpages for FreeBSD and Linux systems:

- Section 1: General Commands
- Section 2: System Calls
- Section 3: C Library Functions
- Section 4: Special Files

So Section 1 is for general commands (a.k.a. shell commands). If I wanted to refer you to the manual page for the `find` command I would write it like this: find(1). This tells you that there is a manual page for `find` in section 1 of the manpages.

If I wanted to refer to the manual page for the `getpid` system call I would write it like this: getpid(2). This tells you that there is a manual page for `getpid` in section 2 of the manpages.

Why do manpages need multiple sections? Because a command may be available in more than one section, ie. available as both a shell command and a system call.

Take stat(1) and stat(2) as an example.

In order to access other sections of the manpages you can specify it like this on the command line:

```
$ man 2 getpid
$ man 3 malloc
$ man find # same as man 1 find
```

This nomenclature was not invented for this book, it's a convention that's used everywhere [1] when referring to the manpages. So it's a good idea to learn it now and get comfortable with seeing it.

Processes: The Atoms of Unix

Processes are the building blocks of a Unix system. Why? Because **any code that is executed happens inside a process**.

For example, when you launch `ruby` from the command line a new process is created for your code. When your code is finished that process exits.

```
$ ruby -e "p Time.now"
```

The same is true for all code running on your system. You know that MySQL server that's always running? That's running in its own process. The e-reader software you're using right now? That's running in its own process. The email client that's desperately trying to tell you you have new messages? You should ignore it by the way and keep reading! It also runs in its own process.

1. http://en.wikipedia.org/wiki/Man_page#Usage

Things start to get interesting when you realize that one process can spawn and manage many others. We'll be taking a look at that over the course of this book.

Chapter 3

Processes Have IDs

Every process running on your system has a unique process identifier, hereby referred to as 'pid'.

The pid doesn't say anything about the process itself, it's simply a sequential numeric label. This is how the kernel sees your process: as a number.

Here's how we can inspect the current pid in a ruby program. Fire up `irb` and try this:

```
# This line will print the pid of the current ruby process. This might be an
# irb process, a rake process, a rails server, or just a plain ruby script.
puts Process.pid
```

A pid is a simple, generic representation of a process. Since it's not tied to any aspect of the content of the process it can be understood from any programming language and with simple tools. We'll see below how we can use the pid to trace the process details using different utilities.

Cross Referencing

To get a full picture, we can use ps(1) to cross-reference our pid with what the kernel is seeing. Leaving your `irb` session open run the following command at a terminal:

```
$ ps -p <pid-of-irb-process>
```

That command should show a process called 'irb' with a pid matching what was printed in the irb session.

In the Real World

Just knowing the pid isn't all that useful in itself. So where is it used?

A common place you'll find pids in the real world is in log files. When you have multiple processes logging to one file it's imperative that you're able to tell which log line comes from which process. Including the pid in each line solves that problem.

Including the pid also allows you to cross reference information with the OS, through the use of commands like top(1) or lsof(8). Here's some sample output from the Spyglass server booting up. The first square brackets of each line denote the pid where the log line is coming from.

```
[58550] [Spyglass::Server] Listening on port 4545
[58550] [Spyglass::Lookout] Received incoming connection
[58557] [Spyglass::Master] Loaded the app
[58557] [Spyglass::Master] Spawned 4 workers. Babysitting now...
[58558] [Spyglass::Worker] Received connection
```

System Calls

Ruby's `Process.pid` maps to getpid(2).

There is also a global variable that holds the value of the current pid. You can access it with `$$`.

Ruby inherits this behaviour from other languages before it (both Perl and bash support `$$`), however *I avoid it when possible*. Typing out `Process.pid` in full is much more expressive of your intent than the dollar-dollar variable, and less likely to confuse those who haven't seen the dollar-dollar before.

Chapter 4

Processes Have Parents

Every process running on your system has a parent process. Each process knows its parent process identifier (hereby referred to as 'ppid').

In the majority of cases the parent process for a given process is the process that invoked it. For example, you're an OSX user who starts up Terminal.app and lands in a bash prompt. Since everything is a process that action started a new Terminal.app process, which in turn started a bash process.

The parent of that new bash process will be the Terminal.app process. If you then invoke ls(1) from the bash prompt, the parent of that `ls` process will be the `bash` process. You get the picture.

Since the kernel deals only in pids there is a way to get the pid of the current parent process. Here's how it's done in Ruby:

```
# Notice that this is only one character different from getting the
# pid of the current process.
puts Process.ppid
```

Cross Referencing

Leaving your `irb` session open run the following command at a terminal:

```
$ ps -p <ppid-of-irb-process>
```

That command should show a process called 'bash' (or 'zsh' or whatever) with a pid that matches the one that was printed in your `irb` session.

In the Real World

There aren't a ton of uses for the ppid in the real world. It can be important when detecting daemon processes, something covered in a later chapter.

System Calls

Ruby's `Process.ppid` maps to getppid(2).

Chapter 5

Processes Have File Descriptors

In much the same way as pids represent running processes, file descriptors represent open files.

Everything is a File

A part of the Unix philosophy: in the land of Unix 'everything is a file'. This means that devices are treated as files, sockets and pipes are treated as files, and files are treated as files.

Since all of these things are treated as files **I'm going to use the word 'resource' when I'm talking about files in a general sense** (including devices, pipes, sockets, etc.) and **I'll use the word 'file' when I mean the classical definition** (a file on the file system).

Descriptors Represent Resources

Any time that you open a resource in a running process it is assigned a file descriptor number. File descriptors are NOT shared between unrelated processes, they live and die with the process they are bound to, just as any open resources for a process are

closed when it exits. There are special semantics for file descriptor sharing when you fork a process, more on that later.

In Ruby, open resources are represented by the `IO` class. Any `IO` object can have an associated file descriptor number. Use `IO#fileno` to get access to it.

```
passwd = File.open('/etc/passwd')
puts passwd.fileno
```

outputs:

```
3
```

Any resource that your process opens gets a unique number identifying it. This is how the kernel keeps track of any resources that your process is using.

What happens when we have multiple resources open?

```ruby
passwd = File.open('/etc/passwd')
puts passwd.fileno

hosts = File.open('/etc/hosts')
puts hosts.fileno

# Close the open passwd file. The frees up its file descriptor
# number to be used by the next opened resource.
passwd.close

null = File.open('/dev/null')
puts null.fileno
```

outputs:

```
3
4
3
```

There are two key takeaways from this example.

1. File descriptor numbers are assigned the lowest unused value. The first file we opened, `passwd`, got file descriptor #3, the next open file got #4 because #3 was already in use.

2. Once a resource is closed its file descriptor number becomes available again. Once we closed the `passwd` file its file descriptor number became available again. So when we opened the file at `dev/null` it was assigned the lowest unused value, which was then #3.

It's important to note that file descriptors keep track of open resources only. Closed resources are not given a file descriptor number.

Stepping back to the kernel's viewpoint again this makes a lot of sense. Once a resource is closed it no longer needs to interact with the hardware layer so the kernel can stop keeping track of it.

Given the above, file descriptors are sometimes called 'open file descriptors'. This is a bit of misnomer since there is no such thing as a 'closed file descriptor'. In fact, trying to read the file descriptor number from a closed resource will raise an exception:

```
passwd = File.open('/etc/passwd')
puts passwd.fileno
passwd.close
puts passwd.fileno
```

outputs:

```
3
-e:4:in `fileno': closed stream (IOError)
```

You may have noticed that when we open a file and ask for its file descriptor number the lowest value we get is 3. What happened to 0, 1, and 2?

Standard Streams

Every Unix process comes with three open resources. These are your standard input (STDIN), standard output (STDOUT), and standard error (STDERR) resources.

These standard resources exist for a very important reason that we take for granted today. STDIN provides a generic way to read input from keyboard devices or pipes, STDOUT and STDERR provide generic ways to write output to monitors, files, printers, etc. This was one of the innovations of Unix.

Before STDIN existed your program had to include a keyboard driver for all the keyboards it wanted to support! And if it wanted to print something to the screen it had to know how to manipulate the pixels required to do so. So let's all be thankful for standard streams.

```
puts STDIN.fileno
puts STDOUT.fileno
puts STDERR.fileno
```

outputs:

```
0
1
2
```

That's where those first 3 file descriptor numbers went to.

In the Real World

File descriptors are at the core of network programming using sockets, pipes, etc. and are also at the core of any file system operations.

Hence, they are used by every running process and are at the core of most of the interesting stuff you can do with a computer. You'll see many more examples of how to use them in the following chapters or in the attached Spyglass project.

System Calls

Many methods on Ruby's `IO` class map to system calls of the same name. These include open(2), close(2), read(2), write(2), pipe(2), fsync(2), stat(2), among others.

Chapter 6

Processes Have Resource Limits

In the last chapter we looked at the fact that open resources are represented by file descriptors. You may have noticed that when resources aren't being closed the file descriptor numbers continue to increase. It begs the question: how many file descriptors can one process have?

The answer depends on your system configuration, but the important point is **there are some resource limits imposed on a process by the kernel**.

Finding the Limits

We'll continue on the subject of file descriptors. Using Ruby we can ask directly for the maximum number of allowed file descriptors:

```
p Process.getrlimit(:NOFILE)
```

On my machine this snippet outputs:

```
[2560, 9223372036854775807]
```

We used a method called `Process.getrlimit` and asked for the maximum number of open files using the symbol `:NOFILE`. It returned a two-element Array.

The first element in the Array is the *soft limit* for the number of file descriptors, the second element in the Array is the *hard limit* for the number of file descriptors.

Soft Limits vs. Hard Limits

What's the difference? Glad you asked. The *soft limit isn't really a limit*. Meaning that if you exceed the soft limit (in this case by opening more than 2560 resources at once) an exception *will* be raised, but you can always change that limit if you want to.

> Note that the hard limit on my system for the number of file descriptors is a ridiculously large integer. Is it even possible to open that many? Likely not, I'm sure you'd run into hardware constraints before that many resources could be opened at once.
>
> On my system that number actually represents infinity. It's repeated in the constant `Process::RLIM_INFINITY`. Try comparing those two values to be sure. So, on my system, I can effectively open as many resources as I'd like, once I bump the soft limit for my needs.

So any process is able to change its own soft limit, but what about the hard limit? Typically that can only be done by a superuser. However, your process is also able to bump the hard limit assuming it has the required permissions. If you're interested in changing the limits at a system-wide level then start by having a look at sysctl(8).

Bumping the Soft Limit

Let's go ahead and bump the soft limit for the current process:

```
Process.setrlimit(:NOFILE, 4096)
p Process.getrlimit(:NOFILE)
```

outputs:

```
[4096, 4096]
```

You can see that we set a new limit for the number of open files, and upon asking for that limit again *both* the hard limit and the soft limit were set to the new value 4096.

We can optionally pass a third argument to `Process.setrlimit` specifying a new hard limit as well, assuming we have the permissions to do so. Note that lowering the hard limit, as we did in that last snippet, is irreversible: once it comes down it won't go back up.

The following example is a common way to raise the soft limit of a system resource to be equal with the hard limit, the maximum allowed value.

```
Process.setrlimit(:NOFILE, Process.getrlimit(:NOFILE)[1])
```

Exceeding the Limit

Note that exceeding the soft limit will raise `Errno::EMFILE`:

```
# Set the maximum number of open files to 3. We know this
# will be maxed out because the standard streams occupy
# the first three file descriptors.
Process.setrlimit(:NOFILE, 3)

File.open('/dev/null')
```

outputs:

```
Errno::EMFILE: Too many open files - /dev/null
```

Other Resources

You can use these same methods to check and modify limits on other system resources. Some common ones are:

```ruby
# The maximum number of simultaneous processes
# allowed for the current user.
Process.getrlimit(:NPROC)

# The largest size file that may be created.
Process.getrlimit(:FSIZE)

# The maximum size of the stack segment of the
# process.
Process.getrlimit(:STACK)
```

Have a look at the documentation [1] for `Process.getrlimit` for a full listing of the available options.

In the Real World

Needing to modify limits for system resources isn't a common need for most programs. However, for some specialized tools this can be very important.

One use case is any process needing to handle thousands of simultaneous network connections. An example of this is the httperf(1) http performance tool. A command like `httperf --hog --server www --num-conn 5000` will ask httperf(1) to create 5000 concurrent connections. Obviously this will be a problem on my system due to its default soft limit, so httperf(1) will need to bump its soft limit before it can properly do its testing.

1. http://www.ruby-doc.org/core-1.9.3/Process.html#method-c-setrlimit

Another real world use case for limiting system resources is a situation where you execute third-party code and need to keep it within certain constraints. You could set limits for the processes running that code and revoke the permissions required to change them, hence ensuring that they don't use more resources than you allow for them.

System Calls

Ruby's `Process.getrlimit` and `Process.setrlimit` map to getrlimit(2) and setrlimit(2), respectively.

Chapter 7

Processes Have an Environment

Environment, in this sense, refers to what's known as 'environment variables'. Environment variables are key-value pairs that hold data for a process.

Every process inherits environment variables from its parent. They are set by a parent process and inherited by its child processes. Environment variables are per-process and are global to each process.

Here's a simple example of setting an environment variable in a `bash` shell, launching a Ruby process, and reading that environment variable.

```
$ MESSAGE='wing it' ruby -e "puts ENV['MESSAGE']"
```

The `VAR=value` syntax is the `bash` way of setting environment variables. The same thing can be accomplished in Ruby using the `ENV` constant.

```
# The same thing, with places reversed!
ENV['MESSAGE'] = 'wing it'
system "echo $MESSAGE"
```

Both of these examples print:

```
wing it
```

In `bash` environment variables are accessed using the syntax: `$VAR`. As you can tell from these few examples environment variables can be used to share state between processes running different languages, `bash` and `ruby` in this case.

It's a hash, right?

Although `ENV` uses the hash-style accessor API it's not actually a `Hash`. For instance, it implements `Enumerable` and some of the `Hash` API, but not all of it. Key methods like `merge` are *not* implemented. So you can do things like `ENV.has_key?`, but don't count on all hash operations working.

```
puts ENV['EDITOR']
puts ENV.has_key?('PATH')
puts ENV.is_a?(Hash)
```

outputs:

```
vim
true
false
```

In the Real World

In the real world environment variables have many uses. Here's a few that are common workflows in the Ruby community:

```
$ RAILS_ENV=production rails server
$ EDITOR=mate bundle open actionpack
$ QUEUE=default rake resque:work
```

Environment variables are often used as a generic way to accept input into a command-line program. Any terminal (on Unix or Windows) already supports them and most programmers are familiar with them. Using environment variables is often less overhead than explicitly parsing command line options.

System Calls

There are no system calls for directly manipulating environment variables, but the C library functions setenv(3) and getenv(3) do the brunt of the work. Also have a look at environ(7) for an overview.

Chapter 8

Processes Have Arguments

Every process has access to a special array called `ARGV`. Other programming languages may implement it slightly differently, but every one has something called 'argv'.

argv is a short form for 'argument vector'. In other words: a vector, or array, of arguments. It holds the arguments that were passed in to the current process on the command line. Here's an example of inspecting `ARGV` and passing in some simple options.

```
$ cat argv.rb
p ARGV
$ ruby argv.rb foo bar -va
["foo", "bar", "-va"]
```

It's an Array!

Unlike the previous chapter, where we learned that `ENV` isn't a `Hash`, `ARGV` is simply an `Array`. You can add elements to it, remove elements from it, change the elements it contains, whatever you like. But if it simply represents the arguments passed in on the command line why would you need to change anything?

Some libraries will read from `ARGV` to parse command line options, for example. You can programmatically change `ARGV` before they have a chance to see it in order to modify the options at runtime.

In the Real World

The most common use case for `ARGV` is probably for accepting filenames into a program. It's very common to write a program that takes one or more filenames as input on the command line and does something useful with them.

The other common use case, as mentioned, is for parsing command line input. There are many Ruby libraries for dealing with command line input. One called `optparse` is available as part of the standard library.

But now that you know how `ARGV` works you can skip that extra overhead for simple command line options and do it by hand. If you just want to support a few flags you can implement them directly as array operations.

```
# did the user request help?
ARGV.include?('--help')
# get the value of the -c option
ARGV.include?('-c') && ARGV[ARGV.index('-c') + 1]
```

Chapter 9

Processes Have Names

Unix processes have very few inherent ways of communicating about their state.

Programmers have worked around this and invented things like logfiles. Logfiles allow processes to communicate anything they want about their state by writing to the filesystem, but this operates at the level of the filesystem rather than being inherent to the process itself.

Similarly, processes can use the network to open sockets and communicate with other processes. But again, that operates at a different level than the process itself, since it relies on the network.

There are two mechanisms that operate at the level of the process itself that can be used to communicate information. One is the process name, the other is exit codes.

Naming Processes

Every process on the system has a name. For example, when you start up an `irb` session that process is given the name 'irb'. The neat thing about process names is that they can be changed at runtime and used as a method of communication.

In Ruby you can access the name of the current process in the `$PROGRAM_NAME` variable. Similarly, you can assign a value to that global variable to change the name of the current process.

```ruby
puts $PROGRAM_NAME

10.downto(1) do |num|
  $PROGRAM_NAME = "Process: #{num}"
  puts $PROGRAM_NAME
end
```

outputs:

```
irb
Process: 10
Process: 9
Process: 8
Process: 7
Process: 6
Process: 5
Process: 4
Process: 3
Process: 2
Process: 1
```

As a fun exercise you can start an `irb` session, print the pid, and change the process name. Then you can use the ps(1) utility to see your changes reflected on the system.

> Unfortunately this global variable (and its mirror `$0`) is the only mechanism provided by Ruby for this feature. There is not a more intent-revealing way to change the name of the current process.

In the Real World

To see an example of how this is used in a real project read through *How Resque Manages Processes* in the appendices.

Chapter 10

Processes Have Exit Codes

When a process comes to an end it has one last chance to make its mark on the world: its exit code. Every process that exits does so with a numeric exit code (0-255) denoting whether it exited successfully or with an error.

Traditionally, a process that exits with **an exit code of 0 is said to be successful**. Any other exit code denotes an error, with different codes pointing to different errors.

Though traditionally they're used to denote different errors, they're really just a channel for communication. All you need to do is handle the different exit codes that a process may exit with in a way that suits your program and you've gotten away from the traditions.

It's usually a good idea to stick with the '0 as success' exit code tradition so that your programs will play nicely with other Unix tools.

How to Exit a Process

There are several ways you can exit a process in Ruby, each for different purposes.

exit

The simplest way to exit a process is using `Kernel#exit`. This is also what happens implicitly when your script ends without an explicit exit statement.

```ruby
# This will exit the program with the success status code (0).
exit
```

```ruby
# You can pass a custom exit code to this method
exit 22
```

```ruby
# When Kernel#exit is invoked, before exiting Ruby invokes any blocks
# defined by Kernel#at_exit.
at_exit { puts 'Last!' }
exit
```

will output:

```
Last!
```

exit!

`Kernel#exit!` is almost exactly the same as `Kernel#exit`, but with two key differences. The first is that it sets an unsuccessful status code by default (1), and the second is that it will not invoke any blocks defined using `Kernel#at_exit`.

```ruby
# This will exit the program with a status code 1.
exit!
```

```
# You can still pass an exit code.
exit! 33
```

```
# This block will never be invoked.
at_exit { puts 'Silence!' }
exit!
```

abort

`Kernel#abort` provides a generic way to exit a process unsuccessfully. `Kernel#abort` will set the exit code to 1 for the current process.

```
# Will exit with exit code 1.
abort
```

```
# You can pass a message to Kernel#abort. This message will be printed
# to STDERR before the process exits.
abort "Something went horribly wrong."
```

```
# Kernel#at_exit blocks are invoked when using Kernel#abort.
at_exit { puts 'Last!' }
abort "Something went horribly wrong."
```

will output:

```
Something went horribly wrong.
Last!
```

raise

A different way to end a process is with an unhandled exception. This is something that you never want to happen in a production environment, but it's almost always happening in development and test environments.

Note that `Kernel#raise`, unlike the previous methods, will not exit the process immediately. It simply raises an exception that *may* be rescued somewhere up the stack. If the exception is not rescued anywhere in the codebase then the unhandled exception will cause the process to exit.

Ending a process this way will still invoke any at_exit handlers and will print the exception message and backtrace to `STDERR`.

```ruby
# Similar to abort, an unhandled exception will set the exit code to 1.
raise 'hell'
```

Chapter 11

Processes Can Fork

Use the fork(2), Luke

Forking is one of the most powerful concepts in Unix programming. The fork(2) system call allows a running process to create new process programmatically. This new process is an exact copy of the original process.

Up until now we've talked about creating processes by launching them from the terminal. We've also mentioned low level operating system processes that create other processes: fork(2) is how they do it.

When forking, the process that initiates the fork(2) is called the "parent", and the newly created process is called the "child".

The child process inherits a copy of all of the memory in use by the parent process, as well as any open file descriptors belonging to the parent process. Let's take a moment to review child processes from the eye of our first three chapters.

Since the child process is an entirely new process, it gets its own unique pid.

The parent of the child process is, obviously, its parent process. So its ppid is set to the pid of the process that initiated the fork(2).

The child process inherits any open file descriptors from the parent at the time of the fork(2). It's given the same map of file descriptor numbers that the parent process has. In this way the two processes can share open files, sockets, etc.

The child process inherits a copy of everything that the parent process has in main memory. In this way a process could load up a large codebase, say a Rails app, that occupies 500MB of main memory. Then this process can fork 2 new child processes. Each of these child processes would effectively have their own copy of that codebase loaded in memory.

The call to `fork` returns near-instantly so we now have 3 processes with each using 500MB of memory. Perfect for when you want to have multiple instances of your application loaded in memory at the same time. Because only one process needs to load the app and forking is fast, this method is faster than loading the app 3 times in separate instances.

The child processes would be free to modify their copy of the memory without affecting what the parent process has in memory. See the next chapter for a discussion of copy-on-write and how it affects memory when forking.

Let's get started with forking in Ruby by looking at a mind-bending example:

```
if fork
  puts "entered the if block"
else
  puts "entered the else block"
end
```

outputs:

```
entered the if block
entered the else block
```

WTF! What's going on here? A call to the `fork` method has taken the once-familiar if construct and turned it on its head. Somehow this piece of code is entering both the if and else block of the if construct!

It's no mystery what's happening here. One call to the `fork` method actually returns twice. Remember that `fork` creates a new process. So it returns once in the calling process (parent) and once in the newly created process (child).

The last example becomes more obvious if we print the pids.

```
puts "parent process pid is #{Process.pid}"

if fork
  puts "entered the if block from #{Process.pid}"
else
  puts "entered the else block from #{Process.pid}"
end
```

outputs:

```
parent process is 21268
entered the if block from 21268
entered the else block from 21282
```

Now it becomes clear that the code in the if block is being executed by the parent process, while the code in the else block is being executed by the child process. The child process will exit after executing its code in the else block, while the parent process will carry on.

Again, there's a rhythm to this beat, and it has to do with the return value of the `fork` method. **In the child process `fork` returns `nil`**. Since `nil` is falsy it executes the code in the else block.

In the parent process `fork` returns the pid of the newly created child process. Since an integer is truthy it executes the code in the if block.

This concept is illustrated nicely by simply printing the return value of a `fork` call.

```
puts fork
```

outputs

```
21423
nil
```

Here we have the two different return values. The first value returned is the pid of the newly created child process; this comes from the parent. The second return value is the nil from the child process.

Multicore Programming?

In a roundabout way, yes. By making new processes it means that your code is able, but not guaranteed, to be distributed across multiple CPU cores.

Given a system with 4 CPUs, if you `fork` 4 new processes then those can be handled each by a separate CPU, giving you multicore concurrency.

However, there's no guarantee that stuff will be happening in parallel. On a busy system it's possible that all 4 of your processes are handled by the same CPU.

fork(2) creates a new process that's a copy of the old process. So if a process is using 500MB of main memory, then it forks, now you have 1GB in main memory.

Do this another ten times and you can quickly exhaust main memory. This is often called a *fork bomb*. Before you turn up the concurrency make sure that you know the consequences.

Using a Block

In the example above we've demonstrated `fork` with an if/else construct. It's also possible, and more common in Ruby code, to use `fork` with a block.

When you pass a block to the `fork` method that block will be executed in the new child process, while the parent process simply skips over it. The child process exits

when it's done executing the block. It does not continue along the same code path as the parent.

```
fork do
  # Code here is only executed in the child process
end

# Code here is only executed in the parent process.
```

In the Real World

Have a look at either of the appendices, or the attached Spyglass project, to see some real-world examples of using fork(2).

System Calls

Ruby's `Kernel#fork` maps to fork(2).

Chapter 12

Orphaned Processes

Out of Control

You may have noticed when running the examples in the last chapter that when child processes are involved, it's no longer possible to control everything from a terminal like we're used to.

When starting a process via a terminal, we normally have only one process writing to `STDOUT`, taking keyboard input, or listening for that Ctrl-C telling it to exit.

But once that process has forked child processes that all becomes a little more difficult. When you press Ctrl-C which process should exit? All of them? Only the parent?

It's good to know about this stuff because it's actually very easy to create orphaned processes:

```
fork do
  5.times do
    sleep 1
    puts "I'm an orphan!"
  end
end

abort "Parent process died..."
```

If you run this program from a terminal you'll notice that since the parent process dies immediately the terminal returns you to the command prompt. At which point, it's overwritten by the `STDOUT` from the child process! Strange things can start to happen when forking processes.

Abandoned Children

What happens to a child process when its parent dies?

The short answer is, *nothing*. That is to say, the operating system doesn't treat child processes any differently than any other processes. So, when the parent process dies the child process continues on; the parent process does not take the child down with it.

Managing Orphans

Can you still manage orphaned processes?

We're getting a bit ahead of ourselves with this question, but it touches on two interesting concepts.

The first is something called daemon processes. Daemon processes are long running processes that are intentionally orphaned and meant to stay running forever. These are covered in detail in a later chapter.

The second interesting bit here is communicating with processes that are not attached to a terminal session. You can do this using something called Unix signals. This is also covered in more detail in a later chapter.

We'll soon talk about how to properly manage and control child processes.

Chapter 13

Processes Are Friendly

Let's take a step back from looking at code for a minute to talk about a higher level concept and how it's handled in different Ruby implementations.

Being CoW Friendly

As mentioned in the forking chapter, fork(2) creates a new child process that's an exact copy of the parent process. This includes a copy of everything the parent process has in memory.

Physically copying all of that data can be considerable overhead, so modern Unix systems employ something called copy-on-write semantics (CoW) to combat this.

As you may have guessed from the name, CoW delays the actual copying of memory until it needs to be written.

So a parent process and a child process will actually share the same physical data in memory until one of them needs to modify it, at which point the memory will be copied so that proper separation between the two processes can be preserved.

```ruby
arr = [1,2,3]

fork do
  # At this point the child process has been initialized.
  # Using CoW this process doesn't need to copy the arr variable,
  # since it hasn't modified any shared values it can continue reading
  # from the same memory location as the parent process.
  p arr
end
```

```ruby
arr = [1,2,3]

fork do
  # At this point the child process has been initialized.
  # Because of CoW the arr variable hasn't been copied yet.
  arr << 4
  # The above line of code modifies the array, so a copy of
  # the array will need to be made for this process before
  # it can modify it. The array in the parent process remains
  # unchanged.
end
```

This is a big win when using fork(2) as it saves on resources. It means that fork(2) is fast since it doesn't need to copy any of the physical memory of the parent. It also means that child processes only get a copy of the data they need, the rest can be shared.

In order for you to have CoW semantics, a Ruby implementation needs to be written in such a way that it doesn't clobber this feature provided by the kernel. Versions of MRI >= 2.0 *are* written in such a way that they respect and preserve these semantics. Versions of MRI <= 1.9 did not preserve the semantics.

But how?

MRI's garbage collector uses a 'mark-and-sweep' algorithm. In a nutshell this means that when the GC is invoked it must traverse the graph of live objects, and for each one the GC must 'mark' it as alive.

In MRI <= 1.9, this 'mark' step was implemented as a modification to that object in memory. So when the GC was invoked right after a `fork`, all live objects were modified, forcing the OS to make copies of all live Ruby objects and foregoing any benefit from CoW semantics.

MRI >= 2.0 still uses a mark-and-sweep GC, but preserves CoW semantics by storing all of the 'marks' in a small data structure in a disparate region of memory. So when the GC runs after a `fork`, this small region of memory must be copied, but the graph of live Ruby objects can be shared between parent and child until your code modifies an object.

What does this mean for you?

If you're building something, or using tools, that depend heavily on fork(2), you should expect much better memory utilization with MRI 2.0 than with earlier versions.

Chapter 14

Processes Can Wait

In the examples of fork(2) up until now we have let the parent process continue on in parallel with the child process. In some cases this led to weird results, such as when the parent process exited before the child process.

That kind of scenario is really only suitable for one use case, fire and forget. It's useful when you want a child process to handle something asynchronously, but the parent process still has its own work to do.

```
message = 'Good Morning'
recipient = 'tree@mybackyard.com'

fork do
  # In this contrived example the parent process forks a child to take
  # care of sending data to the stats collector. Meanwhile the parent
  # process has continued on with its work of sending the actual payload.

  # The parent process doesn't want to be slowed down with this task, and
  # it doesn't matter if this would fail for some reason.
  StatsCollector.record message, recipient
end

# send message to recipient
```

Babysitting

For most other use cases involving fork(2) you'll want some way to keep tabs on your child processes. In Ruby, one technique for this is provided by `Process.wait`. Let's rewrite our orphan-inducing example from the last chapter to perform with less surprises.

```
fork do
  5.times do
    sleep 1
    puts "I am an orphan!"
  end
end

Process.wait
abort "Parent process died..."
```

This time the output will look like:

```
I am an orphan!
I am an orphan!
I am an orphan!
I am an orphan!
I am an orphan!
Parent process died...
```

Not only that, but control will not be returned to the terminal until all of the output has been printed.

So what does `Process.wait` do? `Process.wait` **is a blocking call instructing the parent process to wait for one of its child processes to exit before continuing.**

Process.wait and Cousins

I mentioned something key in that last statement, `Process.wait` blocks until any one of its child processes exit. If you have a parent that's babysitting more than one child process and you're using `Process.wait`, you need to know which one exited. For this, you can use the return value.

`Process.wait` returns the pid of the child that exited. Check it out.

```
# We create 3 child processes.
3.times do
  fork do
    # Each one sleeps for a random amount of number less than 5 seconds.
    sleep rand(5)
  end
end

3.times do
  # We wait for each child process to exit and print the pid that
  # gets returned.
  puts Process.wait
end
```

Communicating with Process.wait2

But wait! `Process.wait` has a cousin called `Process.wait2`!

Why the name confusion? It makes sense once you know that `Process.wait` returns 1 value (pid), but `Process.wait2` returns 2 values (pid, status).

This status can be used as communication between processes via exit codes. In our chapter on Exit Codes we mentioned that you can use exit codes to encode information for other processes. `Process.wait2` gives you direct access to that information.

The `status` returned from `Process.wait2` is an instance of `Process::Status`. It has a lot of useful information attached to it for figuring out exactly how a process exited.

```ruby
# We create 5 child processes.
5.times do
  fork do
    # Each generates a random number. If even they exit
    # with a 111 exit code, otherwise they use a 112 exit code.
    if rand(5).even?
      exit 111
    else
      exit 112
    end
  end
end

5.times do
  # We wait for each of the child processes to exit.
  pid, status = Process.wait2

  # If the child process exited with the 111 exit code
  # then we know they encountered an even number.
  if status.exitstatus == 111
    puts "#{pid} encountered an even number!"
  else
    puts "#{pid} encountered an odd number!"
  end
end
```

Communication between processes without the filesystem or network!

Waiting for Specific Children

But wait! The `Process.wait` cousins have two more cousins. `Process.waitpid` and `Process.waitpid2`.

You can probably guess what these do. They function the same as `Process.wait` and `Process.wait2` except, rather than waiting for *any* child to exit they only wait for a *specific* child to exit, specified by pid.

```
favourite = fork do
  exit 77
end

middle_child = fork do
  abort "I want to be waited on!"
end

pid, status = Process.waitpid2 favourite
puts status.exitstatus
```

Although it appears that `Process.wait` and `Process.waitpid` provide different behaviour don't be fooled! They are actually aliased to the same thing. Both will accept the same arguments and behave the same.

You can pass a pid to `Process.wait` in order to get it to wait for a *specific* child, and you can pass `-1` as the pid to `Process.waitpid` to get it to wait for *any* child process.

The same is true for `Process.wait2` and `Process.waitpid2`.

Just like with `Process.pid` vs. `$$` I think it's important that, as programmers, we use the provided tools to reveal our intent where possible. Although these methods are identical you should use `Process.wait` when you're waiting for any child process and use `Process.waitpid` when you're waiting for a specific process.

Race Conditions

As you look at these simple code examples you may start to wonder about race conditions.

What if the code that handles one exited process is still running when another child process exits? What if I haven't gotten back around to `Process.wait` and another process exits? Let's see:

```ruby
# We create two child processes.
2.times do
  fork do
    # Both processes exit immediately.
    abort "Finished!"
  end
end

# The parent process waits for the first process, then sleeps for 5 seconds.
# In the meantime the second child process has exited and is no
# longer running.
puts Process.wait
sleep 5

# The parent process asks to wait once again, and amazingly enough, the second
# process' exit information has been queued up and is returned here.
puts Process.wait
```

As you can see this technique is free from race conditions. The kernel queues up information about exited processes so that the parent always receives the information in the order that the children exited.

So even if the parent is slow at processing each exited child it will always be able to get the information for each exited child when it's ready for it.

Take note that calling any variant of `Process.wait` when there are no child processes will raise `Errno::ECHILD`. It's always a good idea to keep track of how many child processes you have created so you don't encounter this exception.

In the Real World

The idea of looking in on your child processes is at the core of a common Unix programming pattern. The pattern is sometimes called babysitting processes, master/worker, or preforking.

At the core of this pattern is the concept that you have one process that forks several child processes, for concurrency, and then spends its time looking after them: making sure they are still responsive, reacting if any of them exit, etc.

For example, the Unicorn web server [1] employs this pattern. You tell it how many worker processes you want it to start up for you, 5 for instance.

Then a `unicorn` process will boot up that will `fork` 5 child processes to handle web requests. The parent (or master) process maintains a heartbeat with each child and ensures that all of the child processes stay responsive.

This pattern allows for both concurrency and reliability. Read more about Unicorn in its Appendix at the end of the book.

For an alternative usage of this technique read through the Lookout class in the attached Spyglass project.

System Calls

Ruby's `Process.wait` and cousins map to waitpid(2).

1. http://unicorn.bogomips.org

Chapter 15

Zombie Processes

At the beginning of the last chapter we looked at an example that used a child process to asynchronously handle a task in a fire and forget manner. We need to revisit that example and ensure that we clean up that child process appropriately, lest it become a zombie!

Good Things Come to Those Who wait(2)

In the last chapter I showed that the kernel queues up status information about child processes that have exited. So even if you call `Process.wait` long after the child process has exited its status information is still available. I'm sure you can smell a problem here...

The kernel will retain the status of exited child processes until the parent process requests that status using `Process.wait`. If the parent never requests the status then the kernel can never reap that status information. So creating fire and forget child processes without collecting their status information is a poor use of kernel resources.

If you're not going to wait for a child process to exit using `Process.wait` (or the technique described in the next chapter) then you need to 'detach' that child process. Here's the fire and forget example from last chapter rectified to properly detach the child process:

```
message = 'Good Morning'
recipient = 'tree@mybackyard.com'

pid = fork do
  # In this contrived example the parent process forks a child to take
  # care of sending data to the stats collector. Meanwhile the parent
  # process has continued on with its work of sending the actual payload.

  # The parent process doesn't want to be slowed down with this task, and
  # it doesn't matter if this would fail for some reason.
  StatsCollector.record message, recipient
end

# This line ensures that the process performing the stats collection
# won't become a zombie.
Process.detach(pid)
```

What does `Process.detach` do? It simply spawns a new thread whose sole job is to wait for the child process specified by `pid` to exit. This ensures that the kernel doesn't hang on to any status information we don't need.

What Do Zombies Look Like?

```
# Create a child process that exits after 1 second.
pid = fork { sleep 1 }
# Print its pid.
puts pid
# Put the parent process to sleep so we can inspect the
# process status of the child
sleep 5
```

Running the following command at a terminal, using the pid printed from the last snippet, will print the status of that zombie process. The status should say 'z' or 'Z+', meaning that the process is a zombie.

```
ps -ho pid,state -p [pid of zombie process]
```

In The Real World

Notice that any dead process whose status hasn't been waited on is a zombie process. So *every* child process that dies while its parent is still active will be a zombie, if only for a short time. Once the parent process collects the status from the zombie then it effectively disappears, no longer consuming kernel resources.

It's fairly uncommon to fork child processes in a fire and forget manner, never collecting their status. If work needs to be offloaded in the background it's much more common to do that with a dedicated background queueing system.

That being said there is a Rubygem called spawnling [1] that provides this exact functionality. Besides providing a generic API over processes or threads, it ensures that fire and forget processes are properly detached.

System Calls

There's no system call for `Process.detach` because it's implemented in Ruby simply as a thread and `Process.wait`. The implementation in Rubinius [2] is stark in its simplicity.

1. https://github.com/tra/spawnling
2. https://github.com/rubinius/rubinius/blob/c6e8e33b37601d4a082ddcbbd60a568767074771/kernel/common/process.rb#L377-395

Chapter 16

Processes Can Get Signals

In the last chapter we looked at `Process.wait`. It provides a nice way for a parent process to keep tabs on its child processes. However it is a blocking call: it will not return until a child process dies.

What's a busy parent to do? Not every parent has the luxury of waiting around on their children all day. There is a solution for the busy parent! And it's our introduction to Unix signals.

Trapping SIGCHLD

Let's take a simple example from the last chapter and rewrite it for a busy parent process.

```ruby
child_processes = 3
dead_processes = 0
# We fork 3 child processes.
child_processes.times do
  fork do
    # They sleep for 3 seconds.
    sleep 3
  end
end

# Our parent process will be busy doing some intense mathematics.
# But still wants to know when one of its children exits.

# By trapping the :CHLD signal our process will be notified by the kernel
# when one of its children exits.
trap(:CHLD) do
  # Since Process.wait queues up any data that it has for us we can ask for it
  # here, since we know that one of our child processes has exited.

  puts Process.wait
  dead_processes += 1
  # We exit explicitly once all the child processes are accounted for.
  exit if dead_processes == child_processes
end

# Work it.
loop do
  (Math.sqrt(rand(44)) ** 8).floor
  sleep 1
end
```

SIGCHLD and Concurrency

Before we go on I must mention a caveat. **Signal delivery is unreliable**. By this I mean that if your code is handling a CHLD signal while another child process dies you *may or may not receive a second CHLD signal*.

This can lead to inconsistent results with the code snippet above. Sometimes the timing will be such that things will work out perfectly, and sometimes you'll actually 'miss' an instance of a child process dying.

This behaviour only happens when receiving the same signal several times in quick succession; you can always count on at least one instance of the signal arriving. This same caveat is true for other signals you handle in Ruby; read on to hear more about those.

To properly handle CHLD you must call `Process.wait` in a loop and look for as many dead child processes as are available, since you may have received multiple CHLD signals since entering the signal handler. But....isn't `Process.wait` a blocking call? If there's only one dead child process and I call `Process.wait` again how will I avoid blocking the whole process?

Now we get to the second argument to `Process.wait`. In the last chapter we looked at passing a pid to `Process.wait` as the first argument, but it also takes a second argument, flags. One such flag that can be passed tells the kernel not to block if no child has exited. Just what we need!

There's a constant that represents the value of this flag, `Process::WNOHANG`, and it can be used like so:

```
Process.wait(-1, Process::WNOHANG)
```

Easy enough.

Here's a rewrite of the code snippet from the beginning of this chapter that won't 'miss' any child process deaths:

```ruby
child_processes = 3
dead_processes = 0
# We fork 3 child processes.
child_processes.times do
  fork do
    # They sleep for 3 seconds.
    sleep 3
  end
end

# Sync $stdout so the call to #puts in the CHLD handler isn't
# buffered. Can cause a ThreadError if a signal handler is
# interrupted after calling #puts. Always a good idea to do
# this if your handlers will be doing IO.
$stdout.sync = true

# Our parent process will be busy doing some intense mathematics.
# But still wants to know when one of its children exits.

# By trapping the :CHLD signal our process will be notified by the kernel
# when one of its children exits.
trap(:CHLD) do
  # Since Process.wait queues up any data that it has for us we can ask for it
  # here, since we know that one of our child processes has exited.

  # We loop over a non-blocking Process.wait to ensure that any dead child
  # processes are accounted for.
  begin
    while pid = Process.wait(-1, Process::WNOHANG)
      puts pid
      dead_processes += 1
    end
  rescue Errno::ECHILD
```

```
  end
end

loop do
  # We exit ourself once all the child processes are accounted for.
  exit if dead_processes == child_processes

  sleep 1
end
```

One more thing to remember is that `Process.wait`, even this variant, will raise `Errno::ECHILD` if no child processes exist. Since signals might arrive at any time it's possible for the last CHLD signal to arrive after the previous CHLD handler has already called `Process.wait` twice and gotten the last available status. This asynchronous stuff can be mind-bending. *Any* line of code can be interrupted with a signal. You've been warned!

So you must handle the `Errno::ECHILD` exception in your CHLD signal handler. Also if you don't know how many child processes you are waiting on you should rescue that exception and handle it properly.

Signals Primer

This was our first foray to Unix signals. Signals are asynchronous communication. When a process receives a signal from the kernel it can do one of the following:

1. ignore the signal
2. perform a specified action

3. perform the default action

Where do Signals Come From?

Technically signals are sent by the kernel, just like text messages are sent by a cell phone carrier. But text messages have an original sender, and so do signals. Signals are sent from one process to another process, using the kernel as a middleman.

The original purpose of signals was to specify different ways that a process should be killed. Let's start there.

Let's start up two ruby programs and we'll use one to kill the other.

> For these examples we won't use `irb` because it defines its own signal handlers that get in the way of our demonstrations. Instead we'll just use the `ruby` program itself.
>
> Give this a try: launch the `ruby` program without any arguments. Enter some code. Hit Ctrl-D.
>
> This executes the code that you entered and then exits.

Start up two `ruby` processes using the technique mentioned above and we'll kill one of them using a signal.

1. In the first `ruby` session execute the following code:

```
puts Process.pid
sleep # so that we have time to send it a signal
```

2. In the second `ruby` session issue the following command to kill the first session with a signal:

```
Process.kill(:INT, <pid of first session>)
```

So the second process sent an "INT" signal to the first process, causing it to exit. "INT" is short for "INTERRUPT".

The system default when a process receives this signal is that it should interrupt whatever it's doing and exit immediately.

The Big Picture

Below is a table showing signals commonly supported on Unix systems. Every Unix process will be able to respond to these signals and any signal can be sent to any process.

When naming signals the SIG portion of the name is optional. The Action column in the table describes the default action for each signal:

Term
: means that the process will terminate immediately

Core
means that the process will terminate immediately and dump core (stack trace)

Ign
means that the process will ignore the signal

Stop
means that the process will stop (ie pause)

Cont
means that the process will resume (ie unpause)

```
Signal   Value     Action   Comment
----------------------------------------------------------------------
SIGHUP      1       Term    Hangup detected on controlling terminal
                                    or death of controlling process
SIGINT      2       Term    Interrupt from keyboard
SIGQUIT     3       Core    Quit from keyboard
SIGILL          4       Core    Illegal Instruction
SIGABRT         6       Core    Abort signal from abort(3)
SIGFPE          8       Core    Floating point exception
SIGKILL         9       Term    Kill signal
SIGSEGV        11       Core    Invalid memory reference
SIGPIPE        13       Term    Broken pipe: write to pipe with no readers
SIGALRM        14       Term    Timer signal from alarm(2)
SIGTERM        15       Term    Termination signal
SIGUSR1     30,10,16    Term    User-defined signal 1
SIGUSR2     31,12,17    Term    User-defined signal 2
SIGCHLD     20,17,18    Ign     Child stopped or terminated
SIGCONT     19,18,25    Cont    Continue if stopped
SIGSTOP     17,19,23    Stop    Stop process
SIGTSTP     18,20,24    Stop    Stop typed at tty
SIGTTIN     21,21,26    Stop    tty input for background process
SIGTTOU     22,22,27    Stop    tty output for background process

The signals SIGKILL and SIGSTOP cannot be trapped, blocked, or ignored.
```

This table might seem a bit out of left field, but it gives you a rough idea of what to expect when you send a certain signal to a process. You can see that, by default, most of the signals terminate a process.

It's interesting to note the `SIGUSR1` and `SIGUSR2` signals. These are signals whose action is meant specifically to be defined by your process. We'll see shortly that we're free to

redefine any of the signal actions that we please, but those two signals are meant for your use.

Redefining Signals

Let's go back to our two `ruby` sessions and have some fun.

1. In the first `ruby` session use the following code to redefine the behaviour of the INT signal:

   ```
   puts Process.pid
   trap(:INT) { print "Na na na, you can't get me" }
   sleep # so that we have time to send it a signal
   ```

   Now our process won't exit when it receives the `INT` signal.

2. In the second `ruby` session issue the following command and notice that the first process is taunting us!

   ```
   Process.kill(:INT, <pid of first session>)
   ```

3. You can try using Ctrl-C to kill that first session, and notice that it responds the same!

4. But as the table said there are some signals that cannot be redefined. `SIGKILL` will show that guy who's boss.

```
Process.kill(:KILL, <pid of first session>)
```

Ignoring Signals

1. In the first `ruby` session use the following code:

```
puts Process.pid
trap(:INT, "IGNORE")
sleep # so that we have time to send it a signal
```

2. In the second `ruby` session issue the following command and notice that the first process isn't affected.

```
Process.kill(:INT, <pid of first session>)
```

   The first `ruby` session is unaffected.

Signal Handlers are Global

Signals are a great tool and are the perfect fit for certain situations. But it's good to keep in mind that **trapping a signal is a bit like using a global variable**, you might be overwriting something that some other code depends on. And unlike global variables signal handlers can't be namespaced.

So make sure you read this next section before you go and add signal handlers to all of your open source libraries :)

Being Nice about Redefining Signals

There is a way to preserve handlers defined by other Ruby code, so that your signal handler won't trample any other ones that are already defined. It looks something like this:

```
trap(:INT) { puts 'This is the first signal handler' }

old_handler = trap(:INT) {
  old_handler.call
  puts 'This is the second handler'
  exit
}
sleep 5 # so that we have time to send it a signal
```

Just send it a Ctrl-C to see the effect. Both signal handlers are called.

Now let's see if we can preserve the system default behaviour. Hit the code below with a Ctrl-C.

```
system_handler = trap(:INT) {
  puts 'about to exit!'
  system_handler.call
}
sleep 5 # so that we have time to send it a signal
```

:/ It blew up that time. So we can't preserve the system default behaviour with this technique, but we can preserve other Ruby code handlers that have been defined.

> In terms of best practices your code probably shouldn't define any signal handlers, unless it's a server. As in a long-running process that's booted from the command line. It's very rare that library code should trap a signal.

```
# The 'friendly' method of trapping a signal.

old_handler = trap(:QUIT) {
  # do some cleanup
  puts 'All done!'

  old_handler.call if old_handler.respond_to?(:call)
}
```

This handler for the QUIT signal will preserve any previous QUIT handlers that have been defined. Though this looks 'friendly' it's not generally a good idea. Imagine a scenario where a Ruby server tells its users they can send it a QUIT signal and it will do a graceful shutdown. You tell the users of your library that they can send a QUIT signal and it will draw an ASCII rainbow. Now if a user sends the QUIT signal *both* handlers will be invoked. This violates the expectations of both libraries.

Whether or not you decide to preserve previously defined signal handlers is up to you, just make sure you know why you're doing it. If you simply want to wire up some behaviour to clean up resources before exiting you can use an `at_exit` hook, which we touched on in the chapter about exit codes.

When Can't You Receive Signals?

Your process can receive a signal anytime. That's the beauty of them! They're asynchronous.

Your process can be pulled out of a busy for-loop into a signal handler, or even out of a long `sleep`. Your process can even be pulled from one signal handler to another if it receives one signal while processing another. But, as expected, it will always go back and finish the code in all the handlers that are invoked.

In the Real World

With signals, any process can communicate with any other process on the system, so long as it knows its pid. This makes signals a very powerful communication tool. It's common to send signals from the shell using kill(1).

In the real world signals are mostly used by long running processes like servers and daemons. And for the most part it will be the human users who are sending signals rather than automated programs.

For instance, the Unicorn web server [1] responds to the `INT` signal by killing all of its processes and shutting down immediately. It responds to the `USR2` signal by re-executing itself for a zero-downtime restart. It responds to the `TTIN` signal by incrementing the number of worker processes it has running.

1. http://unicorn.bogomips.org

See the SIGNALS file included with Unicorn [2] for a full list of the signals it supports and how it responds to them.

The memprof project has a interesting example of being a friendly citizen when handling signals [3].

System Calls

Ruby's `Process.kill` maps to kill(2), `Kernel#trap` maps roughly to sigaction(2). signal(7) is also useful.

2. http://unicorn.bogomips.org/SIGNALS.html
3. https://github.com/ice799/memprof/blob/d4bc228aca323b58fea92dbde20c1f8ec36e5386/lib/memprof/signal.rb#L8-16

Chapter 17

Processes Can Communicate

Up until now we've looked at related processes that share memory and share open resources. But what about communicating information between multiple processes?

This is part of a whole field of study called Inter-process communication (IPC for short). There are many different ways to do IPC but I'm going to cover two commonly useful methods: pipes and socket pairs.

Our First Pipe

A pipe is a uni-directional stream of data. In other words you can open a pipe, one process can 'claim' one end of it and another process can 'claim' the other end. Then data can be passed along the pipe but only in one direction. So if one process 'claims' the position of reader, rather than writer, it *will not* be able to write to the pipe. And vice versa.

Before we involve multiple processes let's just look at how to create a pipe and what we get from that:

```
reader, writer = IO.pipe #=> [#<IO:fd 5>, #<IO:fd 6>]
```

`IO.pipe` returns an array with two elements, both of which are `IO` objects. Ruby's amazing IO class [1] is the superclass to `File`, `TCPSocket`, `UDPSocket`, and others. As such, all of these resources have a common interface.

The `IO` objects returned from `IO.pipe` can be thought of something like anonymous files. You can basically treat them the same way you would a `File`. You can call `#read`, `#write`, `#close`, etc. But this object won't respond to `#path` and won't have a location on the filesystem.

Still holding back from bringing in multiple processes let's demonstrate communication with a pipe:

```
reader, writer = IO.pipe
writer.write("Into the pipe I go...")
writer.close
puts reader.read
```

outputs

```
Into the pipe I go...
```

Pretty simple right? Notice that I had to close the writer after I wrote to the pipe? That's because when the reader calls `IO#read` it will continue trying to read data until it sees an EOF (aka. end-of-file marker [2]). This tells the reader that no more data will be available for reading.

So long as the writer is still open the reader might see more data, so it waits. By closing the writer before reading it puts an EOF on the pipe so the reader stops reading after it gets the initial data. If you skip closing the writer then the reader will block and continue trying to read indefinitely.

---

1. http://librelist.com/browser//usp.ruby/2011/9/17/the-ruby-io-class/
2. http://en.wikipedia.org/wiki/End-of-file

Pipes Are One-Way Only

```
reader, writer = IO.pipe
reader.write("Trying to get the reader to write something")
```

outputs

```
 >> reader.write("Trying to get the reader to write something")
IOError: not opened for writing
        from (irb):2:in `write'
        from (irb):2
```

The `IO` objects returned by `IO.pipe` can only be used for uni-directional communication. So the reader can *only* read and the writer can *only* write.

Now let's introduce processes into the mix.

Sharing Pipes

In the chapter on forking I described how open resources are shared, or copied, when a process forks a child. Pipes are considered a resource, they get their own file descriptors and everything, so they are shared with child processes.

Here's a simple example of using a pipe to communicate between a parent and child process. The child indicates to the parent that it has finished an iteration of work by writing to the pipe:

```ruby
reader, writer = IO.pipe

fork do
  reader.close

  10.times do
    # heavy lifting
    writer.puts "Another one bites the dust"
  end
end

writer.close
while message = reader.gets
  $stdout.puts message
end
```

outputs `Another one bites the dust` ten times.

Notice that, like above, the unused ends of the pipe are closed so as not to interfere with EOF being sent. There's actually one more layer when considering EOF now that two processes are involved. Since the file descriptors were copied there's now 4 instances floating around. Since only two of them will be used to communicate the other 2 instances must be closed. Hence the extra instances of closing.

Since the ends of the pipe are `IO` objects we can call any `IO` methods on them, not just `#read` and `#write`. In this example I use `#puts` and `#gets` to read and write a `String` delimited with a newline. I actually used those here to simplify one aspect of pipes: pipes hold a stream of data.

Streams vs. Messages

When I say stream I mean that when writing and reading data to a pipe there's no concept of beginning and end. When working with an IO stream, like pipes or TCP sockets, you write your data to the stream followed by some protocol-specific delimiter. For example, HTTP uses a series of newlines to delimit the headers from the body.

Then when reading data from that IO stream you read it in one chunk at a time, stopping when you come across the delimiter. *That's* why I used `#puts` and `#gets` in the last example: it used a newline as the delimiter for me.

As you may have guessed it's possible to communicate via messages instead of streams. We can't do it with pipe, but we can do it with Unix sockets. Without going into too much detail, Unix sockets are a type of socket that can only communicate on the same physical machine. As such it's much faster than TCP sockets and is a great fit for IPC.

Here's an example where we create a pair of Unix sockets that can communicate via messages:

```
require 'socket'
Socket.pair(:UNIX, :DGRAM, 0) #=> [#<Socket:fd 15>, #<Socket:fd 16>]
```

This creates a pair of UNIX sockets that are already connected to each other. These sockets communicate using datagrams, rather than a stream. In this way you write a *whole* message to one of the sockets and read a *whole* message from the other socket. No delimiters required.

Here's a slightly more complex version of the pipe example where the child process actually waits for the parent to tell it what to work on, then it reports back to the parent once it's finished the work:

```
require 'socket'

child_socket, parent_socket = Socket.pair(:UNIX, :DGRAM, 0)
maxlen = 1000

fork do
  parent_socket.close

  4.times do
    instruction = child_socket.recv(maxlen)
    child_socket.send("#{instruction} accomplished!", 0)
  end
end
child_socket.close

2.times do
  parent_socket.send("Heavy lifting", 0)
end
2.times do
  parent_socket.send("Feather lifting", 0)
end

4.times do
  $stdout.puts parent_socket.recv(maxlen)
end
```

outputs:

```
Heavy lifting accomplished!
Heavy lifting accomplished!
Feather lifting accomplished!
Feather lifting accomplished!
```

So whereas pipes provide uni-directional communication, a socket pair provides bi-directional communication. The parent socket can both read and write to the child socket, and vice versa.

Remote IPC?

IPC implies communication between processes running on the same machine. If you're interested in scaling up from one machine to many machines while still doing something resembling IPC there are a few things to look into. The first one would simply be to communicate via TCP sockets. This option would require more boilerplate code than the others for a non-trivial system. Other plausible solutions would be RPC [3] (remote procedure call), a messaging system like ZeroMQ [4], or the general body of distributed systems [5].

In the Real World

Both pipes and socket pairs are useful abstractions for communicating between processes. They're fast and easy. They're often used as a communication channel instead of a more brute force approach such as a shared database or log file.

---

3. http://en.wikipedia.org/wiki/Remote_procedure_call
4. http://www.zeromq.org/
5. http://en.wikipedia.org/wiki/Distributed_computing

As for which method to use: it depends on your needs. Keep in mind that pipes are uni-directional and socket pairs are bi-directional when weighing your decision.

For a more in-depth example have a look at the Spyglass Master class in the included Spyglass project. It uses a more involved example of the code you saw above where many child processes communicate over a single pipe with their parent process.

System Calls

Ruby's `IO.pipe` maps to pipe(2), `Socket.pair` maps to socketpair(2). `Socket.recv` maps to recv(2) and `Socket.send` maps to send(2).

Chapter 18

Daemon Processes

Daemon processes are processes that run in the background, rather than under the control of a user at a terminal. Common examples of daemon processes are things like web servers, or database servers which will always be running in the background in order to serve requests.

Daemon processes are also at the core of your operating system. There are many processes that are constantly running in the background that keep your system functioning normally. These are things like the window server on a GUI system, printing services or audio services so that your speakers are always ready to play that annoying 'ding' notification.

The First Process

There is one daemon process in particular that has special significance for your operating system. We talked in a previous chapter about every process having a parent process. Can that be true for all processes? What about the very first process on the system?

This is a classic who-created-the-creator kind of problem, and it has a simple answer. When the kernel is bootstrapped it spawns a process called the `init` process. This process has a ppid of `0` and is the 'grandparent of all processes'. It's the first one and it has no ancestor. Its pid is `1`.

Creating Your First Daemon Process

What do we need to get started? Not much. Any process can be made into a daemon process.

Let's look to the rack project [1] for an example here. Rack ships with a `rackup` command to serve applications using different rack supported web servers. Web servers are a great example of a process that will never end; so long as your application is active you'll need a server listening for connections.

The `rackup` command includes an option to daemonize the server and run it in the background. Let's have a look at what that does.

1. http://github.com/rack/rack

Diving into Rack

```
def daemonize_app
  if RUBY_VERSION < "1.9"
    exit if fork
    Process.setsid
    exit if fork
    Dir.chdir "/"
    STDIN.reopen "/dev/null"
    STDOUT.reopen "/dev/null", "a"
    STDERR.reopen "/dev/null", "a"
  else
    Process.daemon
  end
end
```

Lots going on here. Let's first jump to the `else` block. Ruby 1.9.x ships with a method called `Process.daemon` that will daemonize the current process! How convenient!

But don't you want to know how it works under the hood? I knew ya did! The truth is that if you look at the MRI source for `Process.daemon` [2] and stumble through the C code it ends up doing the exact same thing that Rack does in the `if` block above.

So let's continue using that as an example. We'll break down the code line by line.

2. https://github.com/ruby/ruby/blob/c852d76f46a68e2820ofoc3f68c8c67879e79c86/process.c#L4817-4860

Daemonizing a Process, Step by Step

```
exit if fork
```

This line of code makes intelligent use of the return value of the `fork` method. Recall from the forking chapter that `fork` returns twice, once in the parent process and once in the child process. In the parent process it returns the child's pid and in the child process it returns nil.

As always, the return value will be truth-y for the parent and false-y for the child. This means that the parent process will exit, and as we know, orphaned child processes carry on as normal.

If a process is orphaned then what happens when you ask for `Process.ppid`?

This is where knowledge of the `init` process becomes relevant. The ppid of orphaned processes is always `1`. This is the only process that the kernel can be sure is active at all times.

This first step is imperative when creating a daemon because it causes the terminal that invoked this script to think the command is done, returning control to the terminal and taking it out of the equation.

```
Process.setsid
```

Calling `Process.setsid` does three things:

1. The process becomes a session leader of a new session
2. The process becomes the process group leader of a new process group
3. The process has no controlling terminal

To understand exactly what effect these three things have we need to step out of the context of our Rack example for a moment and look a little deeper.

Process Groups and Session Groups

Process groups and session groups are all about job control. By 'job control' I'm referring to the way that processes are handled by the terminal.

We begin with process groups.

Each and every process belongs to a group, and each group has a unique integer id. A process group is just a collection of related processes, typically a parent process and its children. However you can also group your processes arbitrarily by setting their group id using `Process.setpgrp(new_group_id)`.

Have a look at the output from the following snippet.

```
puts Process.getpgrp
puts Process.pid
```

If you ran that code in an `irb` session then those two values will be equal. Typically the process group id will be the same as the pid of the process group leader. The process group leader is the 'originating' process of a terminal command. ie. If you start an `irb` process at the terminal it will become the group leader of a new process group. Any child processes that it creates will be made part of the same process group.

Try out the following example to see that process groups are inherited.

```
puts Process.pid
puts Process.getpgrp

fork {
  puts Process.pid
  puts Process.getpgrp
}
```

You can see that although the child process gets a unique pid it inherits the group id from its parent. So these two processes are part of the same group.

You'll recall that we looked previously at Orphaned Processes. In that section I said that child processes are not given special treatment by the kernel. Exit a parent process and the child will continue on. This is the behaviour when a parent process exits, but the behaviour is a bit different when the parent process is being controlled by a terminal and is killed by a signal.

Consider for a moment: a Ruby script that shells out to a long-running shell command, eg. a long backup script. What happens if you kill the Ruby script with a Ctrl-C?

If you try this out you'll notice that the long-running backup script is not orphaned, it *does not* continue on when its parent is killed. We haven't set up any code to forward the signal from the parent to the child, so how is this done?

The terminal receives the signal and forwards it on to any process in the foreground process group. In this case, both the Ruby script and the long-running shell command would part of the same process group, so they would both be killed by the same signal.

And then session groups...

A session group is one level of abstraction higher up, a collection of process groups. Consider the following shell command:

```
git log | grep shipped | less
```

In this case each command will get its own process group, since each may be creating child processes but none is a child process of another. Even though these commands are not part of the same process group one Ctrl-C will kill them all.

These commands are part of the same session group. Each invocation from the shell gets its own session group. An invocation may be a single command or a string of commands joined by pipes.

Like in the above example, a session group may be attached to a terminal. It might also not be attached to any terminal, as in the case of a daemon.

Again, your terminal handles session groups in a special way: sending a signal to the session leader will forward that signal to all the process groups in that session, which will forward it to all the processes in those process groups. Turtles all the way down ;)

There is a system call for retrieving the current session group id, getsid(2), but Ruby's core library has no interface to it. Using `Process.setsid` will return the id of the new sesssion group it creates, you can store that if you need it.

So, getting back to our Rack example, in the first line a child process was forked and the parent exited. The originating terminal recognized the exit and returned control to the user, but the forked process still has the inherited group id and session id from its parent. At the moment this forked process is neither a session leader nor a group leader.

So the terminal still has a link to our forked process, if it were to send a signal to its session group the forked process would receive it, but we want to be fully detached from a terminal.

`Process.setsid` will make this forked process the leader of a new process group and a new session group. Note that `Process.setsid` will fail in a process that is already a process group leader, it can only be run from child processes.

This new session group does not have a controlling terminal, but technically one could be assigned.

```
exit if fork
```

The forked process that had just become a process group and session group leader forks again and then exits.

This newly forked process is no longer a process group leader nor a session leader. Since the previous session leader had no controlling terminal, and this process is not a

session leader, it's guaranteed that this process can never have a controlling terminal. Terminals can only be assigned to session leaders.

This dance ensures that our process is now fully detached from a controlling terminal and will run to its completion.

```
Dir.chdir "/"
```

This changes the current working directory to the root directory for the system. This isn't strictly necessary but it's an extra step to ensure that current working directory of the daemon doesn't disappear during its execution.

This avoids problems where the directory that the daemon was started from gets deleted or unmounted for any reason.

```
STDIN.reopen "/dev/null"
STDOUT.reopen "/dev/null", "a"
STDERR.reopen "/dev/null", "a"
```

This sets all of the standard streams to go to `/dev/null`, a.k.a. to be ignored. Since the daemon is no longer attached to a terminal session these are of no use anyway. They can't simply be closed because some programs expect them to always be available. Redirecting them to `/dev/null` ensures that they're still available to the program but have no effect.

In the Real World

As mentioned, the `rackup` command ships with a command line option for daemonizing the process. Same goes with any of the popular Ruby web servers.

If you want to dig in to more internals of daemon processes you should look at the `daemons` rubygem [3].

If you think you want to create a daemon process you should ask yourself one basic question: Does this process need to stay responsive forever?

If the answer is no then you probably want to look at a cron job or background job system. If the answer is yes, then you probably have a good candidate for a daemon process.

System Calls

Ruby's `Process.setsid` maps to setsid(2), `Process.getpgrp` maps to getpgrp(2). Other system calls mentioned in this chapter were covered in detail in previous chapters.

3. http://rubygems.org/gems/daemons

Chapter 19

Spawning Terminal Processes

A common interaction in a Ruby program is 'shelling out' from your program to run a command in a terminal. This happens especially when I'm writing a Ruby script to glue together some common commands for myself. There are several ways you can spawn processes to run terminal commands in Ruby.

Before we look at the different ways of 'shelling out' let's look at the mechanism they're all using under the hood.

fork + exec

All of the methods described below are variations on one theme: fork(2) + execve(2).

We've had a good look at fork(2) in previous chapters, but this is our first look at execve(2). It's pretty simple, execve(2) allows you to replace the current process with a different process.

Put another way: execve(2) allows you to transform the current process into any other process. You can take a Ruby process and turn it into a Python process, or an ls(1) process, or another Ruby process.

execve(2) transforms the process and never returns. Once you've transformed your Ruby process into something else you can never come back.

```
exec 'ls', '--help'
```

The fork + exec combo is a common one when spawning new processes. execve(2) is a very powerful and efficient way to transform the current process into another one; the only catch is that your current process is gone. That's where fork(2) comes in handy.

You can use fork(2) to create a new process, then use execve(2) to transform that process into anything you like. Voila! Your current process is still running just as it was before and you were able to spawn any other process that you want to.

If your program depends on the output from the execve(2) call you can use the tools you learned in previous chapters to handle that. `Process.wait` will ensure that your program waits for the child process to finish whatever it's doing so you can get the result back.

File descriptors and exec

At the OS level, a call to execve(2) doesn't close any open file descriptors by default.

However, a call to `exec` in Ruby *will* close all open file descriptors by default (excluding the standard streams).

In other words, the default OS behaviour when you `exec('ls')` would be to give `ls` a copy of any open file descriptors, eg. a database connection. This is rarely what you want, so Ruby's default is to close all open file descriptors before doing an `exec`.

This default behaviour of closing file descriptors on `exec` prevents file descriptor 'leaks'. A leak may happen when you fork + exec to spawn another process that has no need for the file descriptors you currently have open (like your database connections, logfiles, etc.) A leak can waste resources but, even worse, can lead to havoc when you try to close your database connection, only to find that some other process erroneously still has the connection open.

However, you may sometimes want to keep a file descriptor open, to pass an open logfile or live socket to another program being booted via `exec` [1]. You can control this behaviour by passing an options hash to `exec` mapping file descriptor numbers to IO objects, as seen in the following example.

```
hosts = File.open('/etc/hosts')

python_code = %Q[import os; print os.fdopen(#{hosts.fileno}).read()]

# The hash as the last arguments maps any file descriptors that should
# stay open through the exec.
exec 'python', '-c', python_code, {hosts.fileno => hosts}
```

In this example we start up a Ruby program and open the `/etc/hosts` file. Then we `exec` a `python` process and tell it to open the file descriptor number that Ruby received for opening the `/etc/hosts` file. You can see that `python` recognizes this file descriptor (because it was shared via execve(2)) and is able to read from it without having to open the file again.

1. The Unicorn web server uses this exact behavoiur to enable restarts without losing any connections. By passing the open listener socket to the new version of itself through an `exec`, it ensures that the listener socket is never closed during a restart.

Notice the options hash mapping the file descriptor number to the `IO` object. If you remove that hash, the Python program won't be able to open the file descriptor, that declaration keeps it open through the execve(2).

Unlike fork(2), execve(2) does **not** share memory with the newly created process. In the python example above, whatever was allocated in memory for the use of the Ruby program was essentially wiped away when execve(2) was called leaving the python program with a blank slate in terms of memory usage.

Arguments to exec

Notice in all of the examples above I sent an array of arguments to `exec`, rather than passing them as a string? There's a subtle difference to the two argument forms.

Pass a string to `exec` and it will actually start up a shell process and pass the string to the shell to interpret. Pass an array and it will skip the shell and set up the array directly as the `ARGV` to the new process.

Generally you want to avoid passing a string unless you really need to. Pass an array where possible. Passing a string and running code through the shell can raise security concerns. If user input is involved it may be possible for them to inject a malicious command directly in a shell, potentially gaining access to any privileges the current process has. In a case where you want to do something like `exec('ls * | awk '{print($1)}')` you'll have to pass it as a string.

Kernel#system

```
system('ls')
system('ls', '--help')
system('git log | tail -10')
```

The return value of `Kernel#system` reflects the exit code of the terminal command in the most basic way. If the exit code of the terminal command was 0 then it returns `true`, otherwise it returns `false`.

The standard streams of the terminal command are shared with the current process (through the magic of fork(2)), so any output coming from the terminal command should be seen in the same way output is seen from the current process.

Kernel#`

```
`ls`
`ls --help`
%x[git log | tail -10]
```

`Kernel#`` works slightly differently. The value returned is the `STDOUT` of the terminal program collected into a String.

As mentioned, it's using fork(2) under the hood and it doesn't do anything special with `STDERR`, so you can see in the second example that `STDERR` is printed to the screen just as with `Kernel#system`.

`Kernel#`` and `%x[]` do the exact same thing.

Process.spawn

```ruby
# This call will start up the 'rails server' process with the
# RAILS_ENV environment variable set to 'test'.
Process.spawn({'RAILS_ENV' => 'test'}, 'rails server')

# This call will merge STDERR with STDOUT for the duration
# of the 'ls --help' program.
Process.spawn('ls', '--zz', STDERR => STDOUT)
```

`Process.spawn` is a bit different than the others in that it is non-blocking.

If you compare the following two examples you will see that `Kernel#system` will block until the command is finished, whereas `Process.spawn` will return immediately.

```ruby
# Do it the blocking way
system 'sleep 5'

# Do it the non-blocking way
Process.spawn 'sleep 5'

# Do it the blocking way with Process.spawn
# Notice that it returns the pid of the child process
pid = Process.spawn 'sleep 5'
Process.waitpid(pid)
```

The last example in this code block is a really great example of the flexibility of Unix programming. In previous chapters we talked a lot about `Process.wait`, but it was always in the context of forking and then running some Ruby code. You can see from this example that the kernel cares not what you are doing in your process, it will always work the same.

So even though we fork(2) and then run the sleep(1) program (a C program) the kernel still knows how to wait for that process to finish. Not only that, it will be able to properly return the exit code just as was happening in our Ruby programs.

All code looks the same to the kernel; that's what makes it such a flexible system. You can use any programming language to interact with any other programming language, and all will be treated equally.

`Process.spawn` takes many options that allow you to control the behaviour of the child process. I showed a few useful ones in the example above. Consult the official rdoc [2] for an exhaustive list.

IO.popen

```
# This example will return a file descriptor (IO object). Reading from it
# will return what was printed to STDOUT from the shell command.
IO.popen('ls')
```

2. http://www.ruby-doc.org/core-1.9.3/Process.html#method-c-spawn

The most common usage for `IO.popen` is an implementation of Unix pipes in pure Ruby. That's where the 'p' comes from in popen. Underneath it's still doing the fork+exec, but it's also setting up a pipe to communicate with the spawned process. That pipe is passed as the block argument in the block form of `IO.popen`.

```
# An IO object is passed into the block. In this case we open the stream
# for writing, so the stream is set to the STDIN of the spawned process.
#
# If we open the stream for reading (the default) then
# the stream is set to the STDOUT of the spawned process.
IO.popen('less', 'w') { |stream|
  stream.puts "some\ndata"
}
```

With `IO.popen` you have to choose which stream you have access to. You can't access them all at once.

open3

`Open3` allows simultaneous access to the STDIN, STDOUT, and STDERR of a spawned process.

```ruby
# This is available as part of the standard library.
require 'open3'

Open3.popen3('grep', 'data') { |stdin, stdout, stderr|
  stdin.puts "some\ndata"
  stdin.close
  puts stdout.read
}

# Open3 will use Process.spawn when available. Options can be passed to
# Process.spawn like so:
Open3.popen3('ls', '-uhh', :err => :out) { |stdin, stdout, stderr|
  puts stdout.read
}
```

`Open3` acts like a more flexible version of `IO.popen`, for those times when you need it.

In the Real World

All of these methods are common in the Real World. Since they all differ in their behaviour you have to select one based on your needs.

One drawback to all of these methods is that they rely on fork(2). What's wrong with that? Imagine this scenario: You have a big Ruby app that is using hundreds of MB of memory. You need to shell out. If you use any of the methods above you'll incur the cost of forking.

Even if you're shelling out to a simple ls(1) call the kernel will still need to make sure that all of the memory that your Ruby process is using is available for that new ls(1)

process. Why? Because that's the API of fork(2). When you fork(2) the process the kernel doesn't know that you're about to transform that process with an exec(2). You may be forking in order to run Ruby code, in which case you'll need to have all of the memory available.

It's good to keep in mind that fork(2) has a cost, and sometimes it can be a performance bottleneck. What if you need to shell out a lot and don't want to incur the cost of fork(2)?

There are some native Unix system calls for spawning processes without the overhead of fork(2). Unfortunately they don't have support in the Ruby language core library. However, there is a Rubygem that provides a Ruby interface to these system calls. The posix-spawn project [3] provides access to posix_spawn(2), which is available on most Unix systems.

`posix-spawn` mimics the `Process.spawn` API. In fact, most of the options that you pass to `Process.spawn` can also be passed to `POSIX::Spawn.spawn`. So you can keep using the same API and yet reap the benefits of faster, more resource efficient spawning.

At a basic level posix_spawn(2) is a subset of fork(2). Recall the two discerning attributes of a new child process from fork(2): 1) it gets an exact copy of everything that the parent process had in memory, and 2) it gets a copy of all the file descriptors that the parent process had open.

posix_spawn(2) preserves #2, but not #1. That's the big difference between the two. So you can expect a newly spawned process to have access to any of the file descriptors opened by the parent, but it won't share any of the memory. This is what makes posix_spawn(2) faster and more efficient than fork(2). But keep in mind that it also makes it less flexible.

---

3. http://github.com/rtomayko/posix-spawn

System Calls

Ruby's `Kernel#system` maps to system(3), `Kernel#exec` maps to execve(2), `IO.popen` maps to popen(3), posix-spawn uses posix_spawn(2). Ruby controls the 'close-on-exec' behaviour using fcntl(2) with the `FD_CLOEXEC` option.

Chapter 20

Ending

Working with processes in Unix is about two things: abstraction and communication.

Abstraction

The kernel has an extremely abstract (and simple) view of its processes. As programmers we're used to looking at source code as the differentiator between two programs.

We are masters of many programming languages, using each for different purposes. We couldn't possibly write memory-efficient code in a language with a garbage collector, we'll have to use C. But we need objects, let's use C++. On and on.

But if you ask the kernel it all looks the same. In the end, all of our code is compiled down to something simple that the kernel can understand. And when it's working at that level all processes are treated the same. Everything gets its numeric identifier and is given equal access to the resources of the kernel.

What's the point of all this jibber-jabber? Using Unix programming lets you twiddle with these knobs a little bit. It lets you do things that you can't accomplish when working at the programming language level.

Unix programming is programming language agnostic. It lets you interface your Ruby script with a C program, and vice versa. It also lets you reuse its concepts across programming languages. The Unix Programming skills that you get from Ruby will be

just as applicable in Python, or node.js, or C. These are skills that are about programming in general.

Communication

Besides the basic act of creating new processes, almost everything else we talked about was regarding communication. Following the principle of abstraction mentioned above, the kernel provides very abstract ways of communicating between processes.

Using signals any two processes on the system can communicate with each other. By naming your processes you can communicate with any user who is inspecting your program on the command line. Using exit codes you can send success/failure messages to any process that's looking after your own.

Farewell, But Not Goodbye

That's the end! Congratulations for making it here! Believe it or not, you now know more than most programmers about the inner workings of Unix processes.

Now that you know the fundamentals you can go out apply your newfound knowledge to anything that you work on. Things are going to start making more sense for you. And the more you apply your newfound knowledge: the clearer things will become. There's no stopping you now.

And we haven't even talked about networking :) We'll save that one for another edition.

Read the appendices at the end of this book for a look at some popular Ruby projects and how they use Unix processes to be awesome.

If you have any feedback on this book, find an error or build something cool with your newfound knowledge, I'd love to hear it. Send a message to jesse@jstorimer.com.
Happy coding!

Chapter 21

Appendix: How Resque Manages Processes

This section looks at how a popular Ruby job queue, Resque [1], effectively manages processes. Specifically it makes use of fork(2) to manage memory, not for concurrency or speed reasons.

The Architecture

To understand why Resque works the way it does we need a basic understanding of how the system works.

From the README:

> Resque is a Redis-backed library for creating background jobs, placing those jobs on multiple queues, and processing them later.

The component that we're interested in is the Resque worker. Resque workers take care of the 'processing them later' part. The job of a Resque worker is to boot up, load your application environment, then connect to Redis and try to reserve any pending background jobs. When it's able to reserve one such job it works off the job, then goes back to step 1. Simple enough.

1. http://github.com/defunkt/resque#readme

For an application of non-trivial size one Resque worker is not enough. So it's very common to spin up multiple Resque workers in parallel to work off jobs.

Forking for Memory Management

Resque workers employ fork(2) for memory management purposes. Let's have a look at the relevant bit of code (from Resque v1.18.0) eand then dissect it line by line.

```
if @child = fork
  srand # Reseeding
  procline "Forked #{@child} at #{Time.now.to_i}"
  Process.wait(@child)
else
  procline "Processing #{job.queue} since #{Time.now.to_i}"
  perform(job, &block)
  exit! unless @cant_fork
end
```

This bit of code is executed every time Resque works off a job.

If you've read through the Forking chapter then you'll already be familiar with the if/else style here. Otherwise go read it now!

We'll start by looking at the code inside the parent process (ie. inside the if block).

```
srand # Reseeding
```

This line is here simply because of a bug [2] in a certain patchlevel of MRI Ruby 1.8.7.

```
procline "Forked #{@child} at #{Time.now.to_i}"
```

`procline` is Resque's internal way of updating the name of the current process. Remember we noted that you can change the name of the current process by setting `$0` but Ruby doesn't include a method for it?

This is Resque's solution. `procline` sets the name of the current process.

```
Process.wait(@child)
```

If you've read the chapter on `Process.wait` then this line of code should be familiar to you.

The `@child` variable was assigned the value of the `fork` call. So in the parent process that will be the child pid. This line of code tells the parent process to block until the child is finished.

Now we'll look at what happens in the child process.

```
procline "Processing #{job.queue} since #{Time.now.to_i}"
```

Notice that both the `if` and `else` block make a call to procline. Even though these two lines are part of the same logical construct they are being executed in two different processes. Since the process name is process-specific these two calls will set the name for the parent and child process respectively.

---

2. http://redmine.ruby-lang.org/issues/4338

```
perform(job, &block)
```

Here in the child process is where the job is actually 'performed' by Resque.

```
exit! unless @cant_fork
```

Then the child process exits.

Why Bother?

As mentioned in the first paragraph of this chapter, Resque isn't doing this to achieve concurrency or to make things faster. In fact, it adds an extra step to the processing of each job which makes the whole thing slower. So why go to the trouble? Why not just process job after job?

Resque uses fork(2) to ensure that the memory usage of its worker processes don't bloat. Let's review what happens when a Resque worker forks and how that affects the Ruby VM.

You'll recall that fork(2) creates a new process that's an exact copy of the original process. The original process, in this case, has preloaded the application environment and nothing else. So we know that after forking we'll have a new process with just the application environment loaded.

Then the child process will go to the task of working off the job. This is where memory usage can go awry. The background job may require that image files are loaded into main memory for processing, or many ActiveRecord objects are fetched from the

database, or any other operation that requires large amounts of main memory to be used.

Once the child process is finished with the job it exits, which releases all of its memory back to the OS to clean up. Then the original process can resume, once again with only the application environment loaded.

So each time after a job is performed by Resque you end up back at a clean slate in terms of memory usage. This means that memory usage may spike when jobs are being worked on, but it should always come back to that nice baseline.

Doesn't the GC clean up for us?

Well, yes, but it doesn't do a great job. It does an OK job. The truth is that MRI's GC has a hard time releasing memory that it doesn't need anymore.

When the Ruby VM boots up it is allocated a certain block of main memory by the kernel. When it uses up all that it has it needs to ask for another block of main memory from the kernel.

Due to numerous issues with Ruby's GC (naive approach, disk fragmentation) it is rare that the VM is able to release a block of memory back to the kernel. So the memory usage of a Ruby process is likely to grow over time, but not to shrink. Now Resque's approach begins to make sense!

If the Resque worker simply worked off each job as it became available then it wouldn't be able to maintain that nice baseline level of memory usage. As soon as it worked on a job that required lots of main memory then that memory would be stuck with the worker process until it exited.

Even if subsequent jobs needed much less memory Ruby would have a hard time giving that memory back to the kernel. Hence, the worker processes would inevitably get bigger over time. Never shrinking.

Thanks to the power of fork(2) Resque workers are reliable and don't need to be restarted after working a certain number of jobs.

Chapter 22

Appendix: How Unicorn Reaps Worker Processes

Any investigation of Unix Programming in the Ruby language would be remiss without many mentions of the Unicorn web server [1]. Indeed, the project has already been mentioned several times in this book.

What's the big deal? Unicorn is a web server that attempts to push as much responsibility onto the kernel as it can. It uses lots of Unix Programming. The codebase is chock full of Unix Programming techniques.

Not only that, but it's performant and reliable. It's used by lots of big Ruby websites like Github and Shopify.

The point is, if this book has whet your appetite and you want to learn more about Unix Programming in Ruby you should plumb the depths of Unicorn. It may take you several trips into the belly of the mythical beast but you will come out with better understanding and new ideas.

Reaping What?

Before we dive into the code I'd like to provide a bit of context about how Unicorn works. At a very high level Unicorn is a pre-forking web server.

1. http://unicorn.bogomips.org

This means that you boot it up and tell it how many worker processes you would like it to have. It starts by initializing its network sockets and loading your application. Then it uses fork(2) to create the worker processes. It uses the master-worker pattern we mentioned in the chapter on forking.

The Unicorn master process keep a heartbeat on each of its workers and ensures they're not taking too long to process requests. The code below is used when you tell the Unicorn master process to exit. As we covered in chapter (Forking) if a parent process doesn't kill its children before it exits they will continue on without stopping.

So it's important that Unicorn clean up after itself before it exits. The code below is invoked as part of Unicorn's exit procedure. Before invoking this code it will send a `QUIT` signal to each of its worker process, instructing it to exit gracefully.

The code below is used by Unicorn (current as of v4.0.0) to clean up its internal representation of its workers and ensure that they all exited properly.

Let's dive in.

```
# reaps all unreaped workers
def reap_all_workers
  begin
    wpid, status = Process.waitpid2(-1, Process::WNOHANG)
    wpid or return
    if reexec_pid == wpid
      logger.error "reaped #{status.inspect} exec()-ed"
      self.reexec_pid = 0
      self.pid = pid.chomp('.oldbin') if pid
      proc_name 'master'
    else
      worker = WORKERS.delete(wpid) and worker.close rescue nil
      m = "reaped #{status.inspect} worker=#{worker.nr rescue 'unknown'}"
      status.success? ? logger.info(m) : logger.error(m)
    end
  rescue Errno::ECHILD
    break
  end while true
end
```

We'll take it one line at a time:

```
begin
  ...
end while true
```

The first thing that I want to draw your attention to is the fact that the `begin` block that's started on the first line of this method actually starts an endless loop. There are others ways to write endless loops in Ruby, but the important part is to keep in mind

that we are in an endless loop so we'll need a hard `return` or a `break` in order to finish this method.

```
wpid, status = Process.waitpid2(-1, Process::WNOHANG)
```

This line should have some familiarity. We looked at `Process.waitpid2` in the chapter on `Process.wait`.

There we saw that passing a valid pid as the first option would cause the `Process.waitpid` call to wait only for that pid. What happens when you pass `-1` to `Process.waitpid`? We know that there are no processes with a pid less than 1, so...

Passing `-1` waits for *any* child process to exit. It turns out that this is the default option to that method. If you don't specify a pid then it uses `-1` by default. In this case, since the author needed to pass something in for the second argument, the first argument couldn't be left blank, so it was set to the default.

Hey, if you're waiting on any child process why not use `Process.wait2` then? I suspect that the author decided here, and I agree with him, that it was most readable to use a `waitpid` variation when specifying a value for the pid. As mentioned above the value specified is simply the default, but nonetheless it's most salient to use `waitpid` if you're specifying any value for the pid.

Remember `Process::WNOHANG` from before? When using this flag if there are no processes that have exited for us then it will not block and simply return nil.

```
wpid or return
```

This line may look a little odd but it's actually a conditional `return` statement. If `wpid` is `nil` then we know that the last line returned `nil`. This would mean that there are no child processes that have exited returning their status to us.

If this is the case then this method will return and its job is done.

```
    if reexec_pid == wpid
      logger.error "reaped #{status.inspect} exec()-ed"
      self.reexec_pid = 0
      self.pid = pid.chomp('.oldbin') if pid
      proc_name 'master'
```

I don't want to spend much time talking about this bit. The 'reexec' stuff has to do with Unicorn internals, specifically how it handles zero-downtime restarts. Perhaps I can cover that process in a future report.

One thing that I will draw your attention to is the call to `proc_name`. This is similar to the `procline` method from the Resque chapter. Unicorn also has a method for changing the display name of the current process. A critical piece of communication with the user of your software.

```
    else
      worker = WORKERS.delete(wpid) and worker.close rescue nil
```

Unicorn stores a list of currently active worker processes in its `WORKERS` constant. `WORKERS` is a hash where the key is the pid of the worker process and the value is an instance of `Unicorn::Worker`.

So this line removes the worker process from Unicorn's internal tracking list (`WORKERS`) and calls `#close` on the worker instance, which closes its no longer needed heartbeat mechanism.

```
m = "reaped #{status.inspect} worker=#{worker.nr rescue 'unknown'}"
```

These lines craft a log message based on the `status` returned from the `Process.waitpid2` call.

The string is crafted by first inspecting the `status` variable. What does that look like? Something like this:

```
#<Process::Status: pid=32227,exited(0)>
# or
#<Process::Status: pid=32308,signaled(SIGINT=2)>
```

It includes the pid of the ended process, as well as the way it ended. In the first line the process exited itself with an exit code of 0. In the second line the process was killed with a signal, `SIGINT` in this case. So a line like that will be added to the Unicorn log.

The second part of the log line `worker.nr` is Unicorn's internal representation of the worker's number.

```
status.success? ? logger.info(m) : logger.error(m)
```

This line takes the crafted log message and sends it to the logger. It uses the `success?` method on the `status` object to log this message as at the INFO level or the ERROR level.

The `success?` method will only return true in one case, when the process exited with an exit code of 0. If it exited with a different code it will return `false`. If it was killed by a signal, it will return `nil`.

```
    rescue Errno::ECHILD
      break
```

This is part of the top-level `begin` statement in this method. If this exception is raised then the endless loop that is this method `break`s and it will return.

The `Errno::ECHILD` exception will be raised by `Process.waitpid2` (or any of its cousins) if there are no child processes for the current processes. If that happens in this case then it means the job of this method is done! All of the child processes have been reaped. So it returns.

Conclusion

If this bit of code interested you and you want to learn more about Unix Programming in Ruby, Unicorn is a great resource. See the official site at http://unicorn.bogomips.org and go learn!

Chapter 23

Appendix: Preforking Servers

I'm glad you made it this far because this chapter may be the most action-packed in the whole book. Preforking servers bring together a lot of the concepts that are explained in this book into a powerful, highly-efficient approach to solving certain problems.

There's a good chance that you've used either Phusion Passenger [1] or Unicorn [2]. Both of those servers, and Spyglass (the web server included with this book), are examples of preforking servers.

At the core of all these projects is the preforking model. There are a few things about preforking that make it special, here are 3:

1. Efficient use of memory.
2. Efficient load balancing.
3. Efficient sysadminning.

We'll look at each in turn.

1. http://www.modrails.com/
2. http://unicorn.bogomips.org

Efficient use of memory

In the chapter on forking we discussed how fork(2) creates a new process that's an *exact copy* of the calling (parent) process. This includes anything that the parent process had in memory at the time.

Loading a Rails App

On my Macbook Pro loading only Rails 3.1 (no libraries or application code) takes in the neighbourhood of 3 seconds. After loading Rails the process is consuming about 70MB of memory.

Whether or not these numbers are exactly the same on your machine isn't significant for our purposes. I'll be referring to these as a baseline in the following examples.

Preforking uses memory more efficiently than does spawning multiple unrelated processes. For comparison, this is like running Unicorn with 10 worker processes compared to running 10 instances of Mongrel (a non-preforking server).

Let's review what will happen from the standpoint of processes, first looking at Mongrel, then at Unicorn, when we boot up 10 instances of each server.

Many Mongrels

Booting up 10 Mongrel processes in parallel will look about the same as booting up 10 Mongrel processes serially.

When booting them in parallel all 10 processes will be competing for resources from the kernel. Each will be consuming resources to load Rails, and each can be expected to take the customary 3 seconds to boot. In total, that's 30 seconds. On top of that, each process will be consuming 70MB of memory once Rails has been loaded. In total, that's 700MB of memory for 10 processes.

A preforking server can do better.

Many Unicorn

Booting up 10 Unicorn workers will make use of 11 processes. One process will be the master, babysitting the other worker processes, of which there are 10.

When booting Unicorn only one process, the master process, will load Rails. There won't be competition for kernel resources.

The master process will take the customary 3 seconds to load, and forking 10 processes will be more-or-less instantaneous. The master process will be consuming 70MB of memory to load Rails and, thanks to copy-on-write, the child processes should not be using any memory on top of what the master was using.

The truth is that it does take some time to fork a process (it's not instantaneous) and that there is some memory overhead for each child process. These values are negligible compared to the overhead of booting many Mongrels. Preforking wins.

Keep in mind that the benefits of copy-on-write are forfeited if you're running MRI. To reap these benefits you need to be using REE.

Efficient load balancing

I already highlighted the fact that fork(2) creates an *exact copy* of the calling process. This includes any file descriptors that the parent process has open.

The Very Basics of Sockets

Efficient load balancing has a lot to do with how sockets work. Since we're talking about web servers: sockets are important. They're at the very core of networking. As I hinted earlier: sockets and networking are a complex topic, too big to fit into this book. But you need to understand the very basic workflow in order to understand this next part.

Using a socket involves multiple steps: 1) A socket is opened and binds to a unique port, 2) A connection is accepted on that socket using accept(2), and 3) Data can

be read from this connection, written to the connection, and ultimately the connection is closed. The socket stays open, but the connection is closed.

Typically this would happen in the same process. A socket is opened, then the process waits for connections on that socket. The connection is handled, closed, and the loop starts over again.

Preforking servers use a different workflow to let the kernel balance heavy load across the socket. Let's look at how that's done.

In servers like Unicorn and Spyglass the first thing that the master process does is open the socket, before even loading the Rails app. This is the socket that is available for external connections from web clients. But the master process *does not* accept connections. Thanks to the way fork(2) works, when the master process forks worker processes each one gets a copy of the open socket.

This is where the magic happens.

Each worker process has an exact copy of the open socket, and each worker process attempts to accept connections on that socket using accept(2). This is where the kernel takes over and balances load across the 10 copies of the socket. It ensures that one, and only one, process can accept each individual connection. Even under heavy load the kernel ensures that the load is balanced and that only one process handles each connection.

Compare this to how Mongrel achieves load balancing.

Given 10 unrelated processes that aren't sharing a socket each one must bind to a unique port. Now a piece of infrastructure must sit in front of all of the Mongrel processes. It must know which port each Mongrel processes is bound to, and it must do the job of making sure that each Mongrel is handling only one connection at a time and that connections are load balanced properly.

Again, preforking wins both for simplicity and resource efficiency.

Efficient sysadminning

This point is less technical, more human-centric.

As someone administering a preforking server you typically only need to issue commands (usually signals) to the master process. It will handle keeping track of and relaying messages to its worker processes.

When administering many instances of a non-preforking server the sysadmin must keep track of each instance, adminster them separately and ensure that their commands are followed.

Basic Example of a Preforking Server

What follows is some really basic code for a preforking server. It can respond to requests in parallel using multiple processes and will leverage the kernel for load balancing. For a more involved example of a preforking server I suggest you check out the Spyglass source code (next chapter) or the Unicorn source code.

```
require 'socket'

# Open a socket.
socket = TCPServer.open('0.0.0.0', 8080)

# Preload app code.
# require 'config/environment'

# Forward any relevant signals to the child processes.
[:INT, :QUIT].each do |signal|
  Signal.trap(signal) {
    wpids.each { |wpid| Process.kill(signal, wpid) }
  }
end

# For keeping track of child process pids.
wpids = []

5.times {
  wpids << fork do
    loop {
      connection = socket.accept
      connection.puts 'Hello Readers!'
      connection.close
    }
  end
}

Process.waitall
```

You can consume it with something like nc(1) or telnet(1) to see it in action.

```
$ nc localhost 8080
$ telnet localhost 8080
```

Notice that I snuck something new into that one? We haven't seen `Process.waitall` yet, it appeared on the last line of the example code above.

`Process.waitall` is simply a convenience method around `Process.wait`. It runs a loop waiting for all child processes to exit and returns an array of process statuses. Useful when you don't actually want to do anything with the process status info, it just waits for the children to exit.

Chapter 24

Appendix: Spyglass

If you want to know even more about Unix processes then your next stop should be the included Spyglass project. Why? Because it was written specifically to showcase Unix programming concepts.

> If you have a copy of this book but didn't get the included code project, send me an email and I'll hook you up: jesse@jstorimer.com.

The case studies you read are meant to showcase the same thing, but at times they can be dense and hard to read when you're new to Unix programming. Spyglass is meant to bridge that gap.

Spyglass' Architecture

Spyglass is a web server. It opens a socket to the outside world and handles web requests. Spyglass parses HTTP, is Rack-compliant, and is awesome.

Here's a brief summary of how to start a Spyglass server and what happens when it receives an HTTP request.

Booting Spyglass

```
$ spyglass
$ spyglass -p other_port
$ spyglass -h # for help
```

Before a Request Arrives

After it boots, control is passed to `Spyglass::Lookout`. This class DOES NOT preload the Rack application and knows nothing about HTTP, it just waits for a connection. At this point in time `Spyglass` is extremely lightweight, it's nothing more than just an open socket.

Connection is Made

When `Spyglass::Lookout` is notified that a connection has been made it forks a `Spyglass::Master` to actually handle the connection. `Spyglass::Lookout` uses `Process.wait` after forking the master process, so it remains idle until the master exits.

`Spyglass::Master` is responsible for preloading the Rack application and forking/babysitting worker processes. The master process itself doesn't know anything about HTTP parsing or request handling.

The real work is done in `Spyglass::Worker`. It accepts connections using the method outlined in the chapter on preforking, leaning on the kernel for load balancing. Once

it has a connection it parses the HTTP request, calls the Rack app, and writes the response to the client.

Things Get Quiet

So long as there is a steady flow of incoming traffic Spyglass continues to act as a preforking server. If its internal timeout is able to expire without receiving any more incoming requests then the master process, and all its worker processes, exit. Control is returned to `Spyglass::Lookout` and the workflow begins again.

Getting Started

Spyglass is not a production-ready server, so don't rush to start using it for your projects! It's a codebase that's meant to be read. It's heavily commented and formatted documentation is generated with rocco [1].

The best thing to do at this point is enter the `code` directory that comes with this book in your terminal, find the Spyglass codebase, and run `rake read`. This will open up the formatted documentation in your browser for your reading pleasure.

Now go forth and read the code! And may the fork(2) be with you!

1. http://rtomayko.github.com/rocco