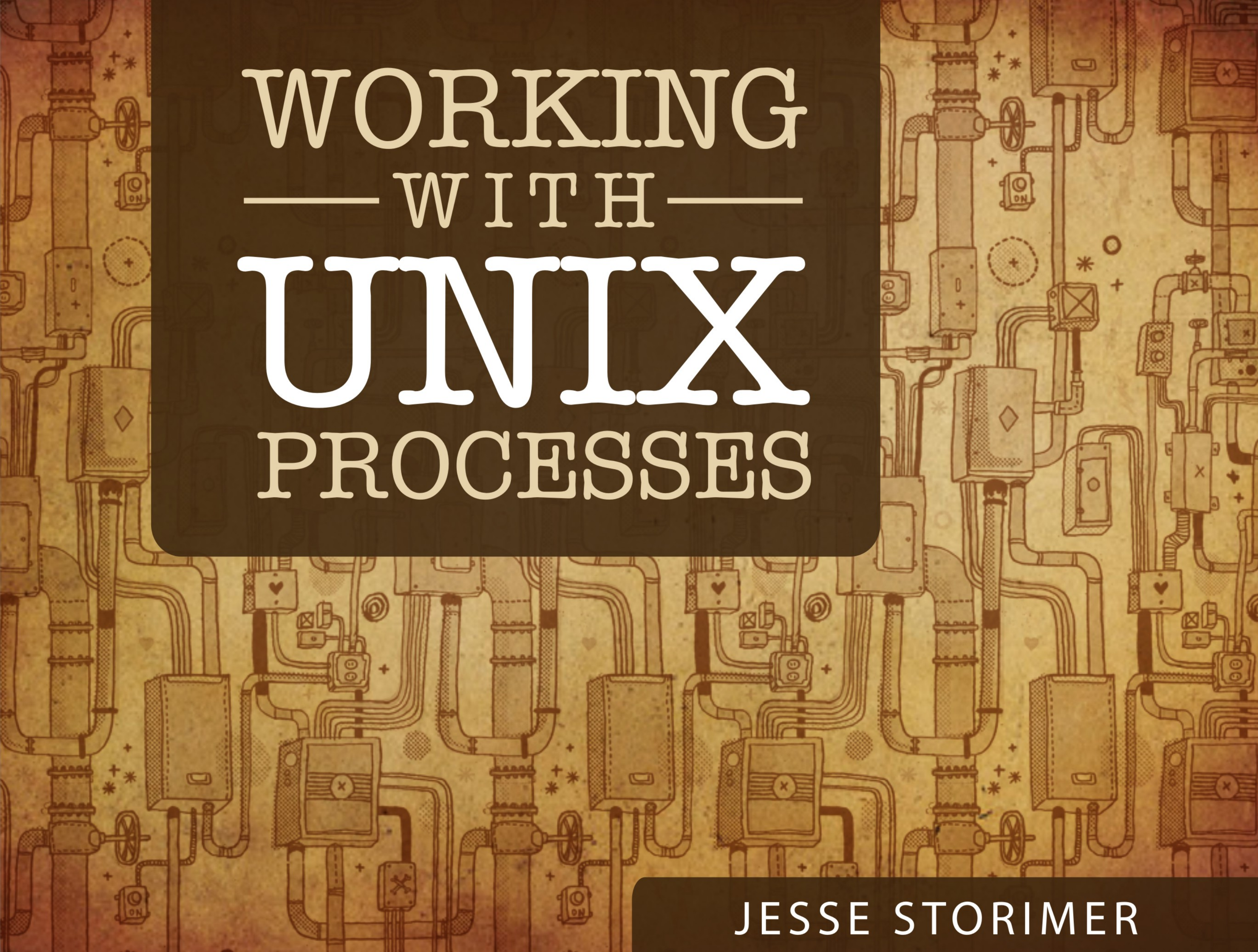
WORKING WITH UNIX PROCESSES

JESSE STORIMER



Working with Unix Processes

Copyright © 2012 Jesse Storimer. All rights reserved. This ebook is licensed for individual use only.

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.

Chapter 17

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.

Diving into Rack

```ruby
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
```

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

```
    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.

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.

2.https://github.com/ruby/ruby/blob/c852d76f46a68e28200f0c3f68c8c67879e79c86/process.c#L4817-4860

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.

```ruby
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.

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

```ruby
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?

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

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.