Table of Contents

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

23. System Administration

Once you start using Ruby, you'll want to use it everywhere. Well, nothing's stopping you. This chapter shows you how to use Ruby in command-line programs that solve general everyday problems. It also demonstrates patterns that you can use to solve your own, more specific everyday problems.

System administration scripts are usually private scripts, disposable or lightly reusable. Ruby scripts are easy to write, so you can get the job done quickly and move on. You won't feel bad if your script is less rigorous than your usual work, and you won't feel invested in a huge program that you only needed once.

Ruby's syntax makes it easy to write, but for system administration, it's the libraries that make Ruby powerful. Most of the recipes in this chapter combine ideas from recipes elsewhere in the book to solve a real-world problem. The most commonly used idea is the `Find.find` technique first covered in Recipe 6.12. Recipes 23.5, 23.6, 23.7, 23.8, and 23.9 all give different twists on this technique.

The major new feature introduced in this chapter is Ruby's standard `etc` library. It lets you query a Unix system's users and groups. It's used in Recipe 23.10 to look up a user's ID given their username. Recipe 23.9 uses it to find a user's home directory and to get the members of Unix groups.

Although these recipes focus mainly on Unix system administration, Ruby is perhaps even more useful for Windows administration. Unix has a wide variety of standard shell tools and an environment that makes it easy to combine them. If Ruby and other high-level languages didn't exist, Unix administrators would still have tools like `find` and `cut`, and they'd use those tools like they did throughout the 1980s. On Windows, though, languages like Ruby are useful even for simple administration tasks: Ruby is easier to use than VBScript or batch files.

If you're trying to administer a Windows machine with Ruby, there are many third-party libraries that provide Ruby hooks into Windows internals: see especially the "win32utils" project at http://rubyforge.org/projects/win32utils/. Another useful library is Ruby's standard `Win32OLE` library, which lets you do things like query Active Directory.

Libraries are also available for the more esoteric parts of Unix systems. See, for instance, Recipe 23.10, which uses the third-party library `sys-proctable` to gain access to the kernel's process table.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

Recipe 23.1. Scripting an External Program

Problem

You want to automatically control an external program that expects to get terminal input from a human user.

Solution

When you're running a program that only needs a single string of input, you can use `IO.popen`, as described in Recipe 20.8. This method runs a command, sends it a string as standard input, and returns the contents of its standard output:

```
def run(command, input='')
  IO.popen(command, 'r+') do |io|
    io.puts input
    io.close_write
    return io.read
   end
end

run 'wc -w', 'How many words are in this string?'      # => "7\n"
```

This technique is commonly used to invoke a command with sudo, which expects the user's password on standard input. This code obtains a user's password and runs a command on his behalf using sudo:

```
print 'Enter your password for sudo: '
sudo_password = gets.chomp
run('sudo apachectl graceful', user_password)
```

Discussion

`IO.popen` is a good way to run noninteractive commands—commands that read all their standard input at once and produce some output. But some programs are interactive; they send prompts to standard output, and expect a human on the other end to respond with more input.

On Unix, you can use Ruby's standard `PTY` and `expect` libraries to spawn a command and impersonate a human on the other end. This code scripts the Unix `passwd` command:

```
require 'expect'
require 'pty'

print 'Old password:'
old_pwd = gets.chomp

print "\nNew password:"
new_pwd = gets.chomp

PTY.spawn('passwd') do |read,write,pid|
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
  write.sync = true
  $expect_verbose = false

  # If 30 seconds pass and the expected text is not found, the
  # response object will be nil.
  read.expect("(current) UNIX password:", 30) do |response|
    write.print old_pwd + "\n" if response
  end

  # You can use regular expressions instead of strings. The code block
  # will give you the regex matches.
  read.expect(/UNIX password: /, 2) do |response, *matches|
    write.print new_pwd + "\n" if response
  end

  # The default value for the timeout is 9999999 seconds
  read.expect("Retype new UNIX password:") do |response|
    write.puts new_pwd + "\n" if response
  end
end
```

The read and write objects in the `PTY#spawn` block are `IO` objects. The `expect` library defines the `IO#expect` method found throughout this example.

See Also

- Recipe 20.8, "Driving an External Process with popen"
- Recipe 21.9, "Reading a Password," shows how to obtain a password without echoing it to the screen

Recipe 23.2. Managing Windows Services

Credit: Bill Froelich

Problem

You want to interact with existing system services on the Windows platform.

Solution

User the `win32-service` library, available as the gem of the same name. Its `Service` module gives you an interface to work with services in Windows 2000 or XP Pro.

You can use this to print a list of the currently running services on your machine:

```
require 'rubygems'
require 'win32/service'
include Win32

puts 'Currently Running Services:'
Service.services do |svc|
  if svc.current_state == 'running'
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
    puts "#{svc.service_name}\t-\t#{svc.display_name}"
  end
end
# Currently Running Services:
# ACPI    -      Microsoft ACPI Driver
# AcrSch2Svc     -       Acronis Scheduler2 Service
# AFD     -      AFD Networking Support Environment
# agp440  -      Intel AGP Bus Filter
# …
```

This command checks whether the DNS client service exists on your machine:

```
Service.exists?('dnscache')                    # => true
```

`Service.status` returns a `Win32ServiceStatus` struct describing the current state of a service:

```
Service.status('dnscache')
# => #<struct Struct::Win32ServiceStatus
#      service_type="share process", current_state="running",
#      controls_accepted=["netbind change", "param change", "stop"],
#      win32_exit_code=0, service_specific_exit_code=0, check_point=0,
#      wait_hint=0, :interactive?=false, pid=1144, service_flags=0>
```

If a service is not currently running, you can start it with `Service.start`:

```
Service.stop('dnscache')
Service.status('dnscache').current_state       # => "stopped"
Service.start('dnscache')
Service.status('dnscache').current_state       # => "running"
```

Discussion

Services are typically accessed using their `service_name` attribute, not by their display name as shown in the Services Control Panel. Fortunately, `Service` provides helpful methods to convert between the two:

```
Service.getdisplayname('dnscache')             # => "DNS Client"
Service.getservicename('DNS Client')           # => "dnscache"
```

In addition to getting information about the status and list of services available, the `win32-service` gem lets you start, pause, and stop services. In the example below, replace the "foo" service with a valid `service_name` that responds to each of the commands.

```
Service.start('foo')
Service.pause('foo')
Service.resume('foo')
Service.stop('foo')
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

You can check whether a service supports `pause` or `resume` by checking the `controls_accepted` member of its `Win32ServiceStatus` struct. As seen below, the `dnscache` command can't be paused or resumed:

```
Service.status('dnscache').controls_accepted
# => ["netbind change", "param change", "stop"]
```

Stopping system services may cause Windows to behave strangely, so be careful.

See Also

- The win32-service library was written by Daniel J. Berger; it's part of his win32utils project (http://rubyforge.org/projects/win32utils/)
- The win32-service APIreference at http://rubyforge.org/docman/view.php/85/29/service.txt; see especially the member list for the `Win32Service` struct yielded by `Service.services`
- You can also use `win32-service` to make your own services; see Recipe 20.2, "Creating a Windows Service"

Recipe 23.3. Running Code as Another User

Problem

While writing a Ruby script that runs as root, you need to take some action on behalf of another user: say, run an external program or create a file.

Solution

Simply set `Process.euid` to the UID of the user. When you're done, set it back to its previous value (that is, root's UID). Here's a method `Process.as_uid` that runs a code block under a different user ID and resets it at the end:

```
module Process
  def as_uid(uid)
    old_euid, old_uid = Process.euid, Process.uid
    Process.euid, Process.uid = uid, uid
    begin
      yield
    ensure
      Process.euid, Process.uid = old_euid, old_uid
    end
  end
  module_function(:as_uid)
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

Discussion

When a Unix process tries to do something that requires special permissions (like access a file), the permissions are checked according to the "effective user ID" of the process. The effective user ID starts out as the user ID you used when you started the process, but if you're root you can change the effective user ID with `Process.euid=`. The operating system will treat you as though you were really that user.

This comes in handy when you're administering a system used by others. When someone asks you for help, you can write a script that impersonates them and runs the commands they don't know how to run. Rather than creating files as root and using `chown` to give them to another user, you can create the files as the other user in the first place.

Here's an example. On my system the account *leonardr* has UID 1000. When run as root, this code will create one directory owned by root and one owned by *leonardr*:

```
Dir.mkdir("as_root")
Process.as_uid(1000) do
  Dir.mkdir("as_leonardr")
  %x{whoami}
end
# => "leonardr\n"
```

Here are the directories:

```
$ ls -ld as_*
drwxr-xr-x 2 leonardr root 4096 Feb 2 13:06 as_leonardr/
drwxr-xr-x 2 root root 4096 Feb 2 13:06 as_root/
```

When you're impersonating another user, your permissions are restricted to what that user can do. I can't remove the `as_root` directory as a nonroot user, because I created it as root:

```
Process.as_uid(1000) do
  Dir.rmdir("as_root")
end
# Errno::EPERM: Operation not permitted - as_root

Dir.rmdir("as_root")                                # => 0
```

On Windows, you can do something like this by splitting your Ruby script into two, and running the second one through `runas.exe`:

```
# script_one.rb
system 'runas /user:frednerk ruby script_two.rb'
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

See Also

- Recipe 6.2, "Checking Your Access to a File"
- If you want to pass in the name of the user to impersonate, instead of their UID, you can adapt the technique shown in Recipe 23.10, "Killing All Processes for a Given User"

Recipe 23.4. Running Periodic Tasks Without cron or at

Problem

You want to write a self-contained Ruby program that performs a task in the background at a certain time, or runs repeatedly at a certain interval.

Solution

Fork off a new process that sleeps until it's time to run the Ruby code.

Here's a program that waits in the background until a certain time, then prints a message:

```
#!/usr/bin/ruby
# lunchtime.rb

def background_run_at(time)
  fork do
    sleep(1) until Time.now >= time
    yield
  end
end

today = Time.now
noon = Time.local(today.year, today.month, today.day, 12, 0, 0)
raise Exception, "It's already past lunchtime!" if noon < Time.now

background_run_at(noon) { puts "Lunchtime!" }
```

The `fork` command only works on Unix systems. The `win32-process` third-party add on gives Windows a `fork` implementation, but it's more idiomatic to run this code as a Windows service with `win32-service`.

Discussion

With this technique, you can write self-contained Ruby programs that act as though they were spawned by the `at` command. If you want to run a backgrounded code block at a certain interval, the way a cronjob would, then combine `fork` with the technique described in Recipe 3.12.

```
#!/usr/bin/ruby
# reminder.rb
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
def background_every_n_seconds(n)
  fork do
    loop do
      before = Time.now
      yield
      interval = n-(Time.now-before)
      sleep(interval) if interval > 0
    end
  end
end

background_every_n_seconds(15*60) { puts 'Get back to work!' }
```

Forking is the best technique if you want to run a background process *and* a foreground process. If you want a script that immediately returns you to the command prompt when it runs, you might want to use the `Daemonize` module instead; see Recipe 20.1.

See Also

- Both the `win32-process` and the `win32-service` libraries are available at http://rubyforge.org/projects/win32utils/
- Recipe 3.12, "Running a Code Block Periodically"
- Recipe 20.1, "Running a Daemon Process on Unix"

Recipe 23.5. Deleting Files That Match a Regular Expression

Credit: Matthew Palmer

Problem

You have a directory full of files and you need to remove some of them. The patterns you want to match are too complex to represent as file globs, but you can represent them as a regular expression.

Solution

The `Dir.entries` method gives you an array of all files in a directory, and you can iterate over this array with `#each`. A method to delete the files matching a regular expression might look like this:

```
def delete_matching_regexp(dir, regex)
  Dir.entries(dir).each do |name|
    path = File.join(dir, name)
    if name =~ regex
      ftype = File.directory?(path) ? Dir : File
      begin
        ftype.delete(path)
      rescue SystemCallError => e
        $stderr.puts e.message
      end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
    end
  end
end
```

Here's an example. Let's create a bunch of files and directories beneath a temporary directory:

```
require 'fileutils'
tmp_dir = 'tmp_buncha_files'
files = ['A', 'A.txt', 'A.html', 'p.html', 'A.html.bak']
directories = ['text.dir', 'Directory.for.html']

Dir.mkdir(tmp_dir) unless File.directory? tmp_dir
files.each { |f| FileUtils.touch(File.join(tmp_dir,f)) }
directories.each { |d| Dir.mkdir(File.join(tmp_dir, d)) }
```

Now let's delete some of those files and directories. We'll delete a file or directory if its name starts with a capital letter, and if its extension (the string after its last period) is at least four characters long. This corresponds to the regular expression `/^[A-Z].*\.[^.]{4,}$/`:

```
Dir.entries(tmp_dir)
# => [".", "..", "A", "A.txt", "A.html", "p.html", "A.html.bak",
#     "text.dir", "Directory.for.html"]

delete_matching_regexp(tmp_dir, /^[A-Z].*\.[^.]{4,}$/)

Dir.entries(tmp_dir)
# => [".", "..", "A", "A.txt", "p.html", "A.html.bak", "text.dir"]
```

Discussion

Like most good things in Ruby, `Dir.entries` takes a code block. It `yields` every file and subdirectory it finds to that code block. Our particular code block uses the regular expression match operator =~ to match every real file (no subdirectories) against the regular expression, and `File.delete` to remove offending files.

`File.delete` won't delete directories; for that, you need `Directory.delete`. So `delete_ matching_regexp` uses the `File` predicates to check whether a file is a directory. We also have error reporting, to report cases when we don't have permission to delete a file, or a directory isn't empty.

Of course, once we've got this basic "find matching files" thing going, there's no reason why we have to limit ourselves to deleting the matched files. We can move them to somewhere new:

```
def move_matching_regexp(src, dest, regex)
  Dir.entries(dir).each do |name|
    File.rename(File.join(src, name), File.join(dest, name)) if name =~ regex
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
  end
end
```

Or we can append a suffix to them:

```
def append_matching_regexp(dir, suffix, regex)
  Dir.entries(dir).each do |name|
    if name =~ regex
      File.rename(File.join(dir, name), File.join(dir, name+suffix))
    end
  end
end
```

Note the common code in both of those implementations. We can factor it out into yet another method that takes a block:

```
def each_matching_regexp(dir, regex)
  Dir.entries(dir).each { |name| yield name if name =~ regex }
end
```

We no longer have to tell `Dir.each` how to match the files we want; we just need to tell `each_matching_regexp` what to do with them:

```
def append_matching_regexp(dir, suffix, regex)
  each_matching_regexp(dir, regex) do |name|
    File.rename(File.join(dir, name), File.join(dir, name+suffix))
  end
end
```

This is all well and good, but these methods only manipulate files directly beneath the directory you specify. "I've got a whole *tree* full of files I want to get rid of!" I hear you cry. For that, you should use `Find.find` instead of `Dir.each`. Apart from that change, the implementation is nearly identical to `delete_matching_regexp`:

```
def delete_matching_regexp_recursively(dir, regex)
  Find.find(dir) do |path|
    dir, name = File.split(path)
    if name =~ regex
      ftype = File.directory?(path) ? Dir : File
      begin
        ftype.delete(path)
      rescue SystemCallError => e
        $stderr.puts e.message
      end
    end
  end
end
```

If you want to recursively delete the contents of directories that match the regular expression (even if the contents themselves don't match), use `FileUtils.rm_rf` instead of `Dir.delete`.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

See Also

- `Dir.delete` will only remove an empty directory; see Recipe 6.18 for information on how to remove one that's not empty
- Recipe 6.20, "Finding the Files You Want"

Recipe 23.6. Renaming Files in Bulk

Problem

You want to rename a bunch of files programmatically: for instance, to normalize the filename case or to change the extensions.

Solution

Use the `Find` module in the Ruby standard library. Here's a method that renames files according to the results of a code block. It returns a list of files it couldn't rename, because their proposed new name already existed:

```
require 'find'

module Find
  def rename(*paths)
    unrenamable = []
    find(*paths) do |file|
      next unless File.file? file # Skip directories, etc.
      path, name = File.split(file)
      new_name = yield name

      if new_name and new_name != name
        new_path = File.join(path, new_name)
        if File.exists? new_path
          unrenamable << file
        else
          puts "Renaming #{file} to #{new_path}" if $DEBUG
          File.rename(file, new_path)
        end
      end
    end
    return unrenamable
  end
  module_function(:rename)
end
```

This addition to the `Find` module makes it easy to do things like convert all filenames to lowercase. I'll create some dummy files to demonstrate:

```
require 'fileutils'
tmp_dir = 'tmp_files'
Dir.mkdir(tmp_dir)
['CamelCase.rb', 'OLDFILE.TXT', 'OldFile.txt'].each do |f|
  FileUtils.touch(File.join(tmp_dir, f))
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
tmp_dir = File.join(tmp_dir, 'subdir')
Dir.mkdir(tmp_dir)
['i_am_SHOUTING', 'I_AM_SHOUTING'].each do |f|
  FileUtils.touch(File.join(tmp_dir, f))
end
```

Now let's convert these filenames to lowercase:

```
$DEBUG = true
Find.rename('./') { |file| file.downcase }
# Renaming ./tmp_files/subdir/I_AM_SHOUTING to ./tmp_files/subdir/i_am_shouting
# Renaming ./tmp_files/OldFile.txt to ./tmp_files/oldfile.txt
# Renaming ./tmp_files/CamelCase.rb to ./tmp_files/camelcase.rb
# => ["./OldFile.txt", "./dir/i_am_SHOUTING"]
```

Two of the files couldn't be renamed, because `oldfile.txt` and `subdir/i_am_shouting` were already taken.

Let's add a ".txt" extension to all files that have no extension:

```
Find.rename('./') { |file| file + '.txt' unless file.index('.') }
# Renaming ./tmp_files/subdir/i_am_shouting to ./tmp_files/subdir/i_am_shouting.txt
# Renaming ./tmp_files/subdir/i_am_SHOUTING to ./tmp_files/subdir/i_am_SHOUTING.txt #
# => []
```

Discussion

Renaming files in bulk is a very common operation, but there's no standard command-line application to do it because renaming operations are best described algorithmically.

The `Find.rename` method makes several simplifying assumptions. It assumes that you want to rename regular files and not directories. It assumes that you can decide on a new name for a file based solely on its filename, not on its full path. It assumes that you'll handle in some other way the files it couldn't rename.

Another implementation might make different assumptions: it might yield both `path` and `name`, and use autoversioning to guarantee that it can rename every file, although not necessary to the exact filename returned by the code block. It all depends on your needs.

Perhaps the most common renaming operation is modifying the extensions of files. Here's a method that uses `Find.rename` to make this kind of operation easier:

```
module Find
  def change_extensions(extension_mappings, *paths)
    rename(*paths) do |file|
      base, extension = file.split(/(.*)\./)[1..2]
      new_extension = extension
      extension_mappings.each do |re, ext|
        if re.match(extension)
          new_extension = ext
          break
        end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
      end
      "#{base}.#{new_extension}"
    end
  end
  module_function(:change_extensions)
end
```

This code uses `Find.change_extensions` to normalize a collection of images. All JPEG files will be given the extension ".jpg", all PNG files the extension ".png", and all GIF files the extension ".gif".

Again, we'll create some dummy image files to test:

```
tmp_dir = 'tmp_graphics'
Dir.mkdir(tmp_dir)

['my.house.jpeg', 'Construction.Gif', 'DSC1001.JPG', '52.PNG'].each do |f|
  FileUtils.touch(File.join(tmp_dir, f))
end
```

Now, let's rename:

```
Find.change_extensions({/jpe?g/i => 'jpg',
                        /png/i => 'png',
                        /gif/i => 'gif'}, tmp_dir)
# Renaming tmp_graphics/52.PNG to tmp_graphics/52.png
# Renaming tmp_graphics/DSC1001.JPG to tmp_graphics/DSC1001.jpg
# Renaming tmp_graphics/Construction.Gif to tmp_graphics/Construction.gif
# Renaming tmp_graphics/my.house.jpeg to tmp_graphics/my.house.jpg
```

See Also

- Some Unix installations come with a program or Perl script called `rename`, which can do your renaming if you can represent it as a string substitution or a regular expression; you may not need anything else
- Recipe 6.14, "Backing Up to Versioned Filenames"
- Recipe 6.20, "Finding the Files You Want"

Recipe 23.7. Finding Duplicate Files

Problem

You want to find the duplicate files that are taking up all the space on your hard drive.

Solution

The simple solution is to group the files by size and then by their MD5 checksum. Two files are presumed identical if they have the same size and MD5 sum.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

The following program takes a list of directories on the command line, and prints out all sets of duplicate files. You can pass a different code block into `each_set_of_duplicates` for different behavior: for instance, to prompt the user about which of the duplicates to keep and which to delete.

```
#!/usr/bin/ruby
# find_duplicates.rb

require 'find'
require 'digest/md5'

def each_set_of_duplicates(*paths)
  sizes = {}
  Find.find(*paths) do |f|
   (sizes[File.size(f)] ||= []) << f if File.file? f
  end
  sizes.each do |size, files|
    next unless files.size > 1
    md5s = {}
    files.each do |f|
      digest = Digest::MD5.hexdigest(File.read(f))
      (md5s[digest] ||= []) << f
    end
    md5s.each { |sum, files| yield files if files.size > 1 }
  end
end

each_set_of_duplicates(*ARGV) do |f|
  puts "Duplicates: #{f.join(", ")}"
end
```

Discussion

This is one task that can't be handled with a simple `Find.find` code block, because it's trying to figure out which files have certain relationships *to each other*. `Find.find` takes care of walking the file tree, but it would be very inefficient to try to make a single trip through the tree and immediately spit out a set of duplicates. Instead, we group the files by size and then by their MD5 checksum.

The MD5 checksum is a short binary string used as a stand-in for the contents of a file. It's commonly used to verify that a huge file was downloaded without errors. It's not impossible for two different files to have an MD5 sum, but unless someone is deliberately trying to trick you, it's almost impossible to have two files with the same size *and* the same MD5 sum.

Calculating a MD5 sum is very expensive: it means performing a mathematical calculation on the entire contents of the file. Grouping the files by size beforehand greatly reduces the number of sums that must be calculated, but that's still a lot of I/O. Even if two similarly sized files differ in the first byte, the code above will read the entire files.

Here's a different version of the same program that takes an incremental approach like that seen in Recipe 6.10. When it thinks a set of files might contain duplicates, it makes

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

repeated calls to a method called `eliminate_non_duplicates`. The duplicates are yielded and the nonduplicates discarded over the course of these calls.

```
#!/usr/bin/ruby
# find_duplicates2.rb

require 'find'
BLOCK_SIZE = 1024*8

def each_set_of_duplicates(*paths, &block)
  sizes = Hash.new {|h, k| h[k] = [] }
  Find.find(*paths) { |f| sizes[File.size(f)] << f if File.file? f }

  sizes.each_pair do |size, files|
    next unless files.size > 1
    offset = 0
    files = [files]
    while !files.empty? && offset <= size
      files = eliminate_non_duplicates(files, size, offset, &block)
      offset += BLOCK_SIZE
    end
  end
end
```

The method `eliminate_non_duplicates` takes lists of files that might contain duplicates. It reads each file an eight-kilobyte block at a time, and compares just one block of each file. Files whose blocks don't match the corresponding blocks of any other file are discarded; they're not duplicates. All files with the same block are put into a new list of *possible* duplicates, and sent back to `each_set_of_duplicates`.

If two files are not duplicates, `eliminate_non_duplicates` will eventually find a block where they differ. Otherwise, it will eventually read the last block of each file and confirm them as duplicates.

```
def eliminate_non_duplicates(partition, size, offset)
  possible_duplicates = []
  partition.each do |possible_duplicate_set|
    blocks = Hash.new {|h, k| h[k] = [] }
    possible_duplicate_set.each do |f|
      block = open(f, 'rb') do |file|
        file.seek(offset)
        file.read(BLOCK_SIZE)
      end
      blocks[block || ''] << f
    end
    blocks.each_value do |files|
      if files.size > 1
        if offset+BLOCK_SIZE >= size
          # We know these are duplicates.
          yield files
        else
          # We suspect these are duplicates, but we need to compare
          # more blocks of data.
          possible_duplicates << files
        end
      end
    end
  end
 return possible_duplicates
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
each_set_of_duplicates(*ARGV) do |f|
  puts "Duplicates: #{f.join(", ")}"
end
```

This code is more complicated, but in real-world situations, it's considerably faster. Most files of the same size are not duplicates, and it's cheaper to find this out by reading eight kilobytes than by reading many megabytes and then performing two MD5 sums. This solution also eliminates any last possibility that `each_set_of_duplicates` will claim two files are duplicates when they're not.

See Also

- Recipe 6.10, "Comparing Two Files"
- Recipe 6.12, "Walking a Directory Tree"

Recipe 23.8. Automating Backups

Problem

You want to make a dated archive of a directory to burn to CD or otherwise store on backup media.

Solution

This script copies a directory to a timestamped backup. It reuses the `File.versioned_filename` method defined in Recipe 6.14, so you can create multiple backups in the same time period:

```
require 'fileutils'

def backup(from_dir, to_dir, time_format="-%Y%m%d")
  from_path, from_name = File.split(from_dir)
  now = Time.now.strftime(time_format)
  Dir.mkdir(to_dir) unless File.exists? to_dir
  unless File.directory? to_dir
    raise ArgumentError, "Not a directory: #{to_dir}"
  end
  to = File.versioned_filename(File.join(to_dir, from_name + now))
  FileUtils.cp_r(from_dir, to, :preserve=>true)
  return to
end

# This method copied from "Backing Up to Versioned Filenames"
class File
  def File.versioned_filename(base, first_suffix=".0")
    suffix = nil
    filename = base
    while File.exists?(filename)
      suffix = (suffix ? suffix.succ : first_suffix)
      filename = base + suffix
    end
    return filename
  end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
end

# Create a dummy directory
Dir.mkdir('recipes')
# And back it up.
backup('recipes', '/tmp/backup')        # => "/tmp/backup/recipes-20061031"
backup('recipes', '/tmp/backup')        # => "/tmp/backup/recipes-20061031.0"
backup('recipes', '/tmp/backup', '-%Y%m%d-%H.%M.%S')
# => "/tmp/backup/recipes-20061031-20.48.56"
```

Discussion

The `backup` method recursively copies the contents of a directory into another directory, possibly on another filesystem. It uses the time-based scheme you specify along with `versioned_filename` to uniquely name the destination directory.

As written, the `backup` method uses a lot of space: every time you call it, it creates an entirely new copy of every file in the source directory. Fortunately, the technique has many variations. Instead of copying the files, you can make a timestamped tarball with the techniques from Recipe 12.10. You can archive the files to another computer with the techniques from Recipe 14.11 (although to save space, you should use the `rsync` program instead). You could even automatically check your work into a version control system every so often; this works better with text than with binary files.

See Also

- Recipe 6.14, "Backing Up to Versioned Filenames"
- Recipe 12.10, "Compressing and Archiving Files with Gzip and Tar"
- Recipe 14.11, "Copying a File to Another Machine"

Recipe 23.9. Normalizing Ownership and Permissions in User Directories

Problem

You want to make make sure your users' home directories don't contain world-writable directories, directories owned by other users, or other potential security problems.

Solution

Use the `etc` library to look up a user's home directory and UID from the username. Then use `Find.find` to walk the directory trees, and `File` methods to check and modify access to each file.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

We are looking out for any case where one user's home directory can be modified by some other user. Whenever we find such a case, we fix it with a `File.chmod` or `File.chown` call. In this program, the actual calls are commented out, so that you don't accidentally change your permissions when you just want to test out the program.

```
#!/usr/bin/ruby -w
# normalize_homes.rb

require 'etc'
require 'find'
require 'optparse'

def normalize_home(pwd_entry, maximum_perms=0775, dry_run=true)
  uid, home = pwd_entry.uid, pwd_entry.dir
  username = pwd_entry.name

  puts "Scanning #{username}'s home of #{home}."

  Find.find(home) do |f|
  next unless File.exists? f
    stat = File.stat(f)
    file_uid, file_gid, mode = stat.uid, stat.gid, stat.mode
```

The most obvious thing we want to check is whether the user owns every file in their home directory. With occasional exceptions (such as files owned by the web server), a user should own the files in his or her home directory:

```
# Does the user own the file?
if file_uid != uid
  begin
    current_owner = Etc.getpwuid(file_uid).name
  rescue ArgumentError # No such user; just use UID
    current_owner = "uid #{file_uid}"
  end
  puts " CHOWN #{f}"
  puts " Current owner is #{current_owner}, should be #{username}"
  # File.chown(uid, nil, f) unless dry_run
end
```

A less obvious check involves the Unix group that owns the file. A user can let other people work on a file in their home directory by giving ownership to a user group. But you can only give ownership to a group if you're a member of that group. If a user's home directory contains a file owned by a group the user doesn't belong to, something fishy is probably going on.

```
# Does the user belong to the group that owns the file?
begin
  group = Etc.getgrgid(file_gid)
  group_name = group.name
rescue ArgumentError # No such group
  group_name = "gid #{file_gid}"
end
unless group && (group.mem.member?(username) || group.name == username)
  puts " CHGRP #{f}"
  puts " Current group is #{group_name}, and #{username} doesn't belong."
  # File.chown(nil, uid, f) unless dry_run
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

Finally, we'll check each file's permissions and make sure they are no more permissive than the value passed in as `maximum_perms`. The default value of `0775` allows any kind of file except a world-writable file. If `normalize_home` finds a world-writable file, it will flip the world-writable bit and leave the rest of the permissions alone:

```
    # Does the file have more than the maximum allowed permissions?
    perms = mode & 0777                # Drop non-permission bits
    should_be = perms & maximum_perms
    if perms != should_be
      puts " CHMOD #{f}"
      puts " Current perms are #{perms.to_s(8)}, " +
           "should be #{should_be.to_s(8)}"
      # File.chmod(perms & maximum_perms, f) unless dry_run
    end
  end
end
```

All that's left to do is a simple command-line interface to the `normalize_home` method:

```
dry_run = false
opts = OptionParser.new do |opts|
  opts.on("-D", "--dry-run",
          "Display changes to be made, don't make them.") do
    dry_run = true
  end

  opts.on_tail("-h", "--help", "display this help and exit") do
    puts opts
    exit
  end
end
opts.banner = "Usage: #{__FILE__} [--dry-run] username [username2, …]"
opts.parse!(ARGV)

# Make sure all the users exist.
pwd_entries = ARGV.collect { |username| Etc.getpwnam(username) }

# Normalize all given home directories.
pwd_entries.each { |p| normalize_home(p, 0775, dry_run ) }
```

Discussion

Running this script on my home directory shows over 2,500 problems. These are mostly files owned by root, files owned by UIDs that don't exist on my system (these come from tarballs), and world-writable files. Below I give a sample of the embarrassment:

```
$ ruby -D normalize_homes.rb leonardr

Scanning leonardr's home of /home/leonardr.
 CHOWN /home/leonardr/writing/Ruby Cookbook/sys-proctable-0.7.3/proctable.so
  Current owner is root, should be leonardr
 CHGRP /home/leonardr/writing/Ruby Cookbook/sys-proctable-0.7.3/proctable.so
  Current group is root, and leonardr doesn't belong.
…
 CHOWN /home/leonardr/writing/Ruby Cookbook/rubygems-0.8.4/lib/rubygems.rb
  Current owner is uid 501, should be leonardr
 CHGRP /home/leonardr/writing/Ruby Cookbook/rubygems-0.8.4/lib/rubygems.rb
  Current group is gid 501, and leonardr doesn't belong.
…
 CHMOD /home/leonardr/SORT/gogol-home-2002/mail
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
    Current perms are 722, should be 720
…
```

Running the script as root (and with the `File.chmod` and `File.chown` calls uncommented) fixes all the problems.

You can run the script as yourself to check your own home directory, and it'll fix permission problems on files you own. But if a file is owned by someone else, you can't take it back just because it's in your home directory—that's part of the problem with having a file owned by someone else in your home directory.

As usual with system administration scripts, `normalize.homes.rb` is only a starting point. You'll probably need to adapt this program to your specific purposes. For instance, you may want to leave certain files alone, especially files owned by root (who can modify anyone's home directory anyway) or by system processes such as the web server (usually user `apache, httpd`, or `nobody`).

See Also

- Recipe 2.6, "Converting Between Numeric Bases"
- Recipe 6.2, "Checking Your Access to a File"
- Recipe 6.3, "Changing the Permissions on a File"
- Recipe 6.12, "Walking a Directory Tree"

Recipe 23.10. Killing All Processes for a Given User

Problem

You want an easy way to kill all the running processes of a user whose processes get out of control.

Solution

You can send a Unix signal (including the deadly SIGTERM or the even deadlier SIGKILL) from Ruby with the `Process.kill` method. But how to get the list of processes for a given user? The simplest way is to call out to the unix `ps` command and parse the output. Running `ps -u#{username}` gives us the processes for a particular user.

```
#!/usr/bin/ruby -w
# banish.rb
def signal_all(username, signal)
  lookup_uid(username)
  killed = 0
  %x{ps -u#{username}}.each_with_index do |proc, i|
    next if i == 0                    # Skip the header provided by ps
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

```
    pid = proc.split[0].to_i
    begin
      Process.kill(signal, pid)
    rescue SystemCallError => e
      raise e unless e.errno == Errno::ESRCH
    end
    killed += 1
  end
  return killed
end
```

There are a couple things to look out for here.

- `ps` dumps a big error message if we pass in the name of a nonexistent user. It would look better if we could handle that error ourselves. That's what the call to `lookup_uid` will do.
- `ps` prints out a header as its first line. We want to skip that line because it doesn't represent a process.
- Killing a process also kills all of its children. This can be a problem if the child process shows up later in the `ps` list: killing it again will raise a `SystemCallError`. We deal with that possibility by catching and ignoring that particular `SystemCallError`. We still count the process as "killed," though.

Here's the implementation of `lookup_id`:

```
def lookup_uid(username)
  require 'etc'
  begin
     user = Etc.getpwnam(username)
  rescue ArgumentError
    raise ArgumentError, "No such user: #{username}"
  end
  return user.uid
end
```

Now all that remains is the command-line interface:

```
require 'optparse'
signal = "SIGHUP"
opts = OptionParser.new do |opts|
  opts.banner = "Usage: #{__FILE__} [-9] [USERNAME]"
   opts.on("-9", "--with-extreme-prejudice",
           "Send an uncatchable kill signal.") { signal = "SIGKILL" }
end
opts.parse!(ARGV)

if ARGV.size != 1
  $stderr.puts opts.banner
  exit
end

username = ARGV[0]
if username == "root"
  $stderr.puts "Sorry, killing all of root's processes would bring down the system."
  exit
end
puts "Killed #{signal_all(username, signal)} process(es)."
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19

As root, you can do some serious damage with this tool:

```
$ ./banish.rb peon
5 process(es) killed
```

Discussion

The main problem with `banish.rb` as written is that it depends on an external program. What's worse, it depends on parsing the human-readable output of an external program. For a quick script this is fine, but this would be more reliable as a self-contained program.

You can get a Ruby interface to the Unix process table by installing the `sysproctable` library. This makes it easy to treat the list of currently running processes as a Ruby data structure. Here's an alternate implementation of `signal_all` that uses `sys-proctable` instead of invoking a separate program. Note that, unlike the other implementation, this one actually uses the return value of `lookup_uid`:

```
def signal_all(username, signal)
  uid = lookup_uid(username)
  require 'sys/proctable'
  killed = 0
  Sys::ProcTable.ps.each do |proc|
    if proc.uid == uid
      begin
        Process.kill(signal, proc.pid)
      rescue SystemCallError => e
        raise e unless e.errno == Errno::ESRCH
      end
      killed += 1
    end
  end
  return killed
end
```

See Also

- `sys-proctable` is in the RAA at http://raa.ruby-lang.org/project/sys-proctable/; it's one of the sysutils packages: see http://rubyforge.org/projects/sysutils for the others
- To write an equivalent program for Windows, you'd either use WMIthrough Ruby's `win32ole` standard library, or install a native binary of GNU's `ps` and use `win32-process`

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly Media, Inc.
Print Publication Date: 2006/07/19