Teaching Operating Systems with Ruby

Robert Sheehan
Department of Computer Science
University of Auckland
New Zealand
0064 9 3737599
r.sheehan@auckland.ac.nz

ABSTRACT

Dynamic languages have regained enormous popularity in recent years. One of the principal dynamic programming languages, Ruby, has been used as the language for assignment work and the presentation of concepts in an introductory Operating Systems course. This was a strange choice for a systems course but there were several good reasons for the choice including the ease with which Ruby provides access to Unix commands and system calls. After some initial problems, the change has been very successful and demonstrates that even in the core courses of a Computer Science curriculum dynamic programming languages have benefits.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education.*

D.4.9 [**Operating Systems**]: Systems Programs and Utilities – *Command and control languages.*

General Terms

Human Factors, Languages.

Keywords

Ruby, scripting language, Operating Systems education.

1. INTRODUCTION

Many students in introductory Operating Systems courses have not been exposed to the languages used to implement or control Operating Systems. This means that students either have to learn a new language to interact with the Operating Systems they study or they are restricted to dealing with simulations or libraries that interface with the Operating System.

If they are to learn a new language the question becomes which one is most appropriate. In recent years students who have been using Java for their other courses may have learnt C for the Operating Systems course.

Another approach is to use a shell scripting language. In the world of Operating System administration many tasks are carried out with scripting languages, ranging from traditional shell script languages such as the Bourne Shell [2] through to more recent general purpose languages Perl [15], Python [8] and Ruby [13].

Ruby is both easy to learn, and very powerful when it comes to interacting with common Operating Systems such as variants of Unix or Microsoft Windows. Moreover Ruby is rapidly becoming known as a useful Web development tool through the Ruby on Rails framework [14]. This means that many students are keen to learn the language. Could such a language be used as the main language of instruction in an Operating Systems course?

1.1 Operating System Course Languages

Traditionally, Operating Systems courses and textbooks have been taught using C [12] or assembly language as the language of demonstration and practice. Some courses and textbooks have used Pascal [6], or Java [11] as these languages became, at least temporarily, the preferred languages for introductory programming courses.

Different languages have been used in Operating Systems courses for different reasons. C and assembly language have been used because these were the languages that Operating Systems were implemented with. This had the advantage that sections of Operating System code could be examined to see exactly how algorithms worked, especially in the ubiquitous Unix Version 6 source-code [7] that was freely available to academic institutions, and more recently in Linux [3]. The sections of code written in assembly language decreased rapidly with the introduction of Unix and this led to a corresponding decline in the use of assembly language in Operating Systems courses.

Textbooks and courses that used Pascal did so because of the clarity with which algorithms could be expressed and the fact that for a while Pascal was the major programming language in academic Computer Science. Pascal was not widely used in the implementation of Operating Systems with the exception of the UCSD Pascal Operating System [4].

Silberschatz, Galvin and Gagne [11] produced a version of their text book with examples and questions in Java. As Java runs on a virtual machine, it seems an odd choice for an Operating System language. The Java programmer is not exposed to the hardware or Operating System the Java Virtual Machine is running on. From a pedagogical perspective Java still makes sense as the language for an Operating System course as it is the most common language in programming courses and it is cross platform. In addition, Java provides many services that are covered in Operating Systems courses, such as threads, concurrency control, security, and distributed systems.

Java does not reveal its interface to the underlying Operating System; this means that rather than directly using Operating System services, such as process creation, interprocess communication, exception handling or file manipulation, a student uses the Java representations which may or may not map closely to the equivalent constructs in the Operating System. If Java is to be used to interface directly with system calls, a package such as Jtux [10] can be used. Without such a package even interfacing with Unix commands is quite intimidating with Java. Figure 1 shows how to do something to each line of output produced by the Unix *ps* command. A scripting language can do this in a much more straightforward way (see Figure 3) – especially on a Unix type system where many commands are provided for system control and administration.

```
import java.io.*;

public class PS {

   public static void main(String[] args) {
      Runtime runtime = Runtime.getRuntime();
      try {
         Process ps_process =
            runtime.exec("ps x");
         BufferedReader in = new BufferedReader(
            new InputStreamReader(
               ps_process.getInputStream()));
         String line;
         while ((line = in.readLine()) != null)
            doSomething(line);
      }
      catch (IOException e);
   }

}
```

Figure 1. Accessing Unix command output from Java.

2. WHY RUBY?

There are several aspects that make Ruby a sensible choice as the main language in an Operating Systems course. Like Java, Ruby works on most common machines and Operating Systems. It also has built-in threads, concurrency control and a simple distributed computing environment. Each of these will be considered briefly in the following sections. Ruby also has a useful exception handling mechanism which avoids the need to explicitly propagate error codes back up through the calling chain and error handling is an important topic in Operating System design.

Ruby is sometimes referred to as a reflective, object-oriented programming language. It is commonly mentioned along with other so-called scripting languages awk, Perl, Python and PHP, but arguably has a nicer syntax and more consistent style than those other languages. Being a scripting language it can be used to manipulate files in sophisticated ways. It has regular expressions built-in to the language which makes many administrative tasks usually performed by awk or Perl very convenient. As Ruby is developed mostly on Linux systems, Unix-like features are conveniently available from the standard Ruby commands and libraries. Because of its flexible nature, Ruby is also ideal at producing domain specific languages to solve particular Operating System problems.

2.1 Threads

User-level threads are provided with all implementations of Ruby. This has the advantage of providing threads on Operating Systems that don't support native threads. This can be used to advantage. E.g., the students in one session of the course were given an assignment to compare Ruby's user-level threads with POSIX threads. The students had to measure how long it took to construct, start and terminate a Ruby thread versus the same times for Pthreads. Similarly they had to compare context-switching times.

The greatest disadvantage of user-level threads is the blocking problem. Normally when one user-level thread in a process blocks for IO, all threads are blocked as the Operating System kernel only schedules the process. Ruby solves this problem with a form of jacketing.

2.2 Concurrency Control

Another important component of Operating Systems courses is a discussion of concurrency control. The Ruby code for many concurrency constructs such as simple locks, condition variables and monitors, comes with the standard distribution and is relatively easy to understand and extend. Figure 2 shows the standard code for locking a simple lock or mutex. The lock variable `@locked` is an instance variable of the Mutex class.

```
# Attempts to grab the lock and waits
# if it isn't available.

  def lock
    while (Thread.critical = true; @locked)
      @waiting.push Thread.current
      Thread.stop
    end
    @locked = true
    Thread.critical = false
    self
  end
```

Figure 2. Ruby simple lock.

In current versions of Ruby, thread synchronization primitives are normally written using a special method `Thread.critical=`. In Ruby code this looks like a simple assignment but is actually a method call that is passed the parameter *true* or *false*. When passed the value *true*, the method causes the thread scheduler to stop scheduling other threads. It starts scheduling threads again when passed the value *false*. This is a very low-level approach to synchronization and is only supposed to be used to implement higher-level thread synchronization mechanisms, but it is very useful from a pedagogical perspective because it functions in a similar way to disabling interrupts and can be used to demonstrate how such a simple mechanism can underlie more sophisticated approaches. In one assignment students were asked to extend the Mutex class so that every time a thread blocked waiting for the lock to become available a deadlock check was carried out. This was very simple to implement.

2.3 Distributed Computing

Operating Systems are no longer the stand alone installations they once were. It is very important in any Operating Systems course to discuss distributed systems. Ruby has nice distributed classes, and has become very popular as the basis for Web applications with the Ruby on Rails [14] framework. Even without the framework, Ruby provides a simple implementation of the Linda

[5] distributed environment, known as Rinda. It is also possible to use the standard drb (distributed ruby) library which provides a distributed object system, similar to Java's RMI, but simpler to use.

2.4 Scripting Language Advantages

Being interpreted and dynamic, Ruby can be used directly from the keyboard using the interactive Ruby program *irb*. In many situations *irb* can be used in place of the shell as a command-line interpreter. This is useful both to experiment with Operating System commands and to help learn the language.

2.5 Unix Connections

Many Operating Systems courses spend substantial time discussing Unix and its related Operating Systems. Ruby provides a large number of Unix connections.

Ruby allows very simple communication with Operating System commands via the traditional shell back-quote mechanism. Figure 3 shows the one line Ruby equivalent to Figure 1.

```
`ps x`.each { |line| do_something line }
```

Figure 3. Accessing Unix command output from Ruby.

The same mechanism works with Microsoft Windows' commands as well.

Concepts such as *pipes*, process creation with *fork*, and *signals* are directly accessible from Ruby. Figure 4 shows a simple use of a *pipe*.

```
  reader, writer = IO.pipe

  writer.puts "from the pipe"
  output = reader.gets
  puts output
```

Figure 4. Simple use of a pipe.

Actually, this code also works under Windows but the code in Figure 5 which employs a *fork* system call only works on versions of Unix.

```
  reader, writer = IO.pipe

  if fork
    puts "parent sending pid"
    writer.puts Process.pid
  else
    puts "child #{Process.pid}"
    puts "child gets: " + reader.gets
  end
```

Figure 5. Fork and pipe.

The code in Figure 5 produces output like:

```
parent sending pid
child 6628
child gets: 6627
```

Compare Figure 5 with the equivalent C program (Figure 6) in order to see the simplification Ruby provides.

```
  #include <unistd.h>
  #include <stdio.h>

  int main(int argc, char **argv) {
      int pipe_ends[2];

      pipe(pipe_ends);
      FILE *reader = fdopen(pipe_ends[0], "r");
      FILE *writer = fdopen(pipe_ends[1], "w");
      if (fork()) {
          printf("parent sending pid\n");
          fprintf(writer, "%d\n", getpid());
      } else {
          printf("child %d\n", getpid());
          char data[10];
          fscanf(reader, "%s", data);
          printf("child gets: %s\n", data);
      }
  }
```

Figure 6, Fork and pipe, C version

Figure 7 gives an example of Unix commands being manipulated with Ruby code. The line numbers are added for reference. In line 1 the Unix *ps* command is invoked with several options. The text output from this command is then split into lines and passed off to the block of code following the `each` method call. The lines are split in line 2 into two String variables `waiting` and `pid`. If the contents of the `waiting` variable matches the regular expression `pipe` (line 3), a message is written to the standard output describing what is about to happen, and then, on line 5, the corresponding process is sent the SIGKILL signal. This code kills any of the user's processes that are currently waiting on a pipe.

```
1: `ps x -o wchan,pid`.each do |line|
2:   waiting, pid = line.split
3:   if waiting =~ /pipe/
4:     puts "about to kill #{pid}"
5:     Process.kill("KILL", pid.to_i)
6:   end
7: end
```

Figure 7. Unix process manipulation.

2.6 Domain Specific Language

Another useful feature of Ruby is its ability to be modified to become a domain specific language. Apart from the usual ability to implement what looks like a domain specific language using method calls, it is possible to produce syntax which matches language features from other environments. As an example, the code in Figure 8 appears to show a `synchronize` keyword in the language which provides mutually exclusive access to the code after it.

```
  synchronize do
    whatever ...
  end
```

Figure 8. A synchronize block.

In reality `synchronize` is a method which takes a block of code as a parameter. The method is shown in Figure 9. The `yield` calls the block of code passed with the `synchronize` call. This method first calls the `lock` method and then when the lock is gained calls the passed block of code (at the `yield` statement) and is guaranteed to call `unlock` before returning, regardless of any errors that may occur.

```
  def synchronize
    lock
    begin
      yield
    ensure
      unlock
    end
  end
```

Figure 9. The synchronize method.

This flexibility was employed in another assignment where the students had to implement a message passing system inspired by the CSP like communication technique in Erlang [1]. The resulting message passing system looked as though it was part of the Ruby language.

3. ADVANTAGES

There are advantages for both the students and the instructors in using Ruby in an Operating Systems course. The conciseness and power of the language and standard libraries means that many topics can be presented with examples that are easy to comprehend without the need for overwhelming amounts of scaffolding code. One further example will illustrate.

To provide a Rinda server that distributed processes (or Operating Systems) can use to post their own services onto requires five lines of code, see Figure 10.

```
require 'rinda/ring'
require 'rinda/tuplespace'

DRb.start_service
Rinda::RingServer.new(Rinda::TupleSpace.new)
DRb.thread.join
```

Figure 10. Rinda TupleSpace server.

3.1 Advantages For The Students

If students have to learn a new language for an Operating Systems course shouldn't it be C? It can certainly be argued that C or C++ are the ideal languages as most current Operating Systems are implemented with them. However, students who have only been exposed to Java have to spend a considerable amount of time becoming proficient with these. Many Operating Systems courses use Java because of this. Also, it is common to use a scripting language in order to administer Operating Systems; this means that a student might have to learn two languages for an Operating Systems course.

As has been shown, Ruby can fulfil the role of communicating with Unix process and system calls which would normally be performed by C. Additionally, Ruby fits ideally as a scripting language to administer Operating Systems. It is also very easy to learn [9]. Students can pick up enough of the basics to deal with assignments within two weeks.

Very few of our students move on to jobs which require the development of Operating Systems. Some will produce device drivers, but in the current Operating Systems' world these are still very system dependent and almost exclusively require C/C++ programming skills. A larger number of the students doing this course are likely to end up with standard programming or system administration type jobs. For these students Ruby has definite appeal.

In Computer Science programs it is advantageous to be exposed to different types of programming language. Dynamic reflective languages such as Ruby are certainly different from Java or C++. Using Ruby in an Operating Systems course demonstrates many of its advantages to students and shows them ways to achieve results they would otherwise not have experienced.

3.1.1 Level-playing Field

The Operating Systems course is taken by two types of students: degree students who have been using Java for over two years and diploma students who enter the course from a variety of backgrounds. One of the problems with using Java as the programming language for the course was that many of the diploma students were at a great disadvantage. Using Ruby has helped this substantially as all students are learning the language together and facing the same problems.

3.2 Advantages For The Instructors

The flexible nature of Ruby along with its clean interaction with Unix system calls provides new scope for assignments. Over the three years Ruby has been used in the Operating Systems course there have been several assignments given to the students that would have been substantially different (if not very difficult to produce) if Java was the course language. Some of these have already been mentioned.

- a comparison of Unix processes, Pthreads and Ruby threads
- the implementation of a local deadlock detector (once developed this worked transparently in any existing or future Ruby program)
- a distributed deadlock detector via a white-board system
- the implementation of an Erlang-like message passing system with timeouts
- a scheduler based on Unix signals which solved the priority inversion problem
- a file-system that allowed Unix-type hard links between directories
- a low-level file-system simulator built on top of a disk simulator

Some of these assignments would have been possible with standard Java, but some would have required the development of native libraries and the solutions would not have been as simple as they were in Ruby. They would all have been possible with C but at the cost of requiring much more scaffolding. Without the scaffolding the main point of the assignment becomes more apparent.

There are some topics in introductory Operating Systems courses that can only be included in practical assignments by simulations. In particular, memory management and virtual memory assignments have been presented in our course over the years via simple simulators. The students either have to extend the simulators or develop algorithms which are implemented on the simulators.

Because of its flexible and dynamic nature, development in Ruby is faster and (for many programmers) more enjoyable. Thus it is possible to produce larger and more flexible simulations within the time constraints of course preparation.

None of the individual advantages of Ruby are overwhelming but after several years of running the introductory Operating Systems course it has been refreshing to use Ruby as the language of example and presentation. Occasionally students are still presented with pseudo Unix source code with little bits of C, but most algorithms can be represented very clearly with Ruby. One last advantage of Ruby is that many programmers find that it is fun to program in. Even though most instructors may find Operating Systems' topics full of interest this does always not seem to be the perception of an average undergraduate.

4. DISADVANTAGES

No Operating Systems have been (or are likely to be) implemented in Ruby. This means that students cannot use Ruby to directly extend a real Operating System. Instead, a student can deal with many of the concepts of Operating Systems in an easy way and can deal quite closely with process and file system calls, especially when Ruby is run on Unix.

Ruby displays its Unix roots quite clearly. The file and process manipulation commands derive from a Unix model. Most file commands work without difficulty on Microsoft Windows Operating Systems as well; Ruby even converts “/” file separators to whatever the underlying Operating System requires. To provide process and service management on Windows it is necessary to add some specific Ruby Windows libraries that do not come with the default distribution.

The general lack of Ruby documentation in English was a major difficulty when the first version of the course was offered; even though it was much better than just a few years previously. Now, due to Ruby’s fast growth and prominence with the Ruby on Rails framework there are over a dozen recent Ruby titles available. Students can be freely given the electronic version of the first edition of the standard Ruby book “Programming Ruby” [13].

5. STAFF AND STUDENT IMPRESSIONS

How well was Ruby received as the programming language for the Operating Systems course? The instructor found much greater freedom to produce different styles of assignment for the practical aspects of the course. The students in the first year Ruby was used had a mixed reaction to the language. Some of them loved it and some hated it. Part of the problem was that there was not enough support in the early part of the course to learn the language. The students were largely expected to pick up the language by themselves (they were third year students). No laboratory assistants had had experience with the language and so questions were frequently referred to the course instructor. As a result of these problems several changes were made for the second year. Tutorials in the language were implemented and an expanded Ruby handout was prepared for the students. Once again some of the novel aspects of Ruby were used to provide assignments that would have been difficult in other languages. This time the course evaluations showed a greatly reduced number of negative comments about Ruby, and, as they had in the first year, the best students really enjoyed learning the language and using it in the assignments.

Student evaluations for the course initially dipped on the introduction of Ruby but have since risen to be higher than the evaluations when Java was used. Most of the content of the course has remained substantially the same over that period.

It is also worth noting that as the students are only exposed to Ruby for the one semester, many of them do not feel confident with their use of the language. In two of the years that the course has been run with Ruby, the students have been allowed to implement the last assignment in any programming language that was appropriate. Most students chose Java rather than Ruby, even though the Ruby solutions were both simpler, and much shorter.

6. CONCLUSION

With features such as its clearly understandable concurrency constructs, a simple distributed computing environment, flexibility to add features to the language, a close connection to Unix process and file system calls and a dynamic nature for simulations, Ruby can be used as the main language of demonstration and assignment work in an introductory Operating Systems course. The experiment to use Ruby in such a course has proven successful and enjoyable for both staff and students, and suggests a greater flexibility in choosing languages in such courses has merit.

7. REFERENCES

1. Armstrong, J., Virding, R., Wikström, C. and Williams, M. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
2. Arthur, L.J. *UNIX Shell programming*. Wiley, New York, 1990.
3. Bovet, D.P. and Cesati, M. *Understanding the Linux Kernel*. O'Reilly & Associates, Sebastopol, CA, 2003.
4. Bowles, K. *Beginner's Guide for the UCSD Pascal System*. McGraw-Hill, 1980.
5. Carriero, N. and Gelernter, D. Linda in context. *Communications of the ACM, 32* (4). 444 - 458.
6. Deitel, H.M. *Operating Systems*. Addison-Wesley, Reading, Massachusetts, 1990.
7. Lions, J. *Lions' Commentary on Unix*. Peer-To-Peer Communications, 1977.
8. Lutz, M. *Programming Python*. O’Reilly & Associates, Sebastopol, CA, 2001.
9. Pine, C. *Learn to Program*. The Pragmatic Bookshelf, Raleigh, North Carolina, 2006.
10. Rochkind, M.J. Jtux — Java-To-UNIX Package, http://basepath.com/aup/jtux/, 2005.
11. Silberschatz, A., Galvin, P.B. and Gagne, G. *Applied operating system concepts*. John Wiley, New York, 2000.
12. Tanenbaum, A.S. and Woodhull, A. *Operating systems : design and implementation*. Prentice Hall, Upper Saddle River, NJ, 1997.
13. Thomas, D. *Programming Ruby: The Pragmatic Programmer's Guide*. Pragmatic Bookshelf, 2004.
14. Thomas, D. and Heinemeier, D. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006.
15. Wall, L. and Schwartz, R.L. *Programming Perl*. O’Reilly & Associates, Sebastopol, CA, 1991.