Experience Teaching a Semester-Long Inferno Course

Phillip Stanley-Marbell
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

In the spring of 2004, a semester-long course for undergraduates was organized as part of CMU's student-taught StuCo (student college) curriculum. The course covered material ranging from a historical background on Inferno's development, the Limbo programming language and related systems such as Communicating Sequential Processes (CSP), to the implementations of the Inferno emulator and native operating system. This paper details the structure of the course, lessons learned explaining concepts about the Inferno and Plan 9 operating systems and presents examples of questions raised and misconceptions incurred by students during the 12 week course.

1. Introduction

It would be fair to say that Inferno may be considered a research operating system, with the attendant connotations that (1) it was developed by a research group/lab, (2) it contains several interesting ideas (3) it may be regarded by some as unpolished and not quite ready for day-to-day use as one's primary operating system and (4) it is made available complete with source, that it may be studied and extended to fit one's needs.

From a didactic viewpoint, Inferno is particularly interesting since, like its contemporaries and predecessors Squeak [1] and Oberon [2], it is available both as a traditional operating system, executing on many hardware platforms, as well as being available as an application — the Inferno emulator — that can be executed on a variety of host operating systems. Since the system, as seen from within the emulator and native operating systems are largely identical, the emulator can be used as an effective tool for introducing the concepts of, and developing applications for, Inferno. Without requiring dedicated hardware on which to install the system, users of the system can get a first-hand view of working within Inferno. A second motivating factor in using Inferno as a teaching tool is to expose students to what could be argued to be *good programming style*. It can be hoped that by so doing, students would pick up good programming practice by osmosis.

An argument often made within academic circles is that it is difficult for students to link formal models for concurrency, such as Hoare's Communicating Sequential Processes (CSP), the Calculus of Communicating Systems (CCS), the π-calculus and so on, with real world concurrent programming. This assertion is not surprising, since these statements are often made with frameworks such as POSIX threads (Pthreads) [3] in mind. The facilities for concurrent programming in Limbo, with its simplicity of thread creation and the availability of typed language-level channels, are in contrast easily related to formal models. Limbo therefore serves as an ideal vehicle for the introduction of concepts of concurrency in an introductory programming course. It was with these ideas in mind that the course, titled *Concurrent and Distributed Programming with Inferno and Limbo* was designed and offered at CMU in the spring of 2004.

The following section details the topics discussed and questions raised in each of the 15 lectures. The paper concludes in Section 3. with a summary of lessons learned, and a retrospective on improvements that might be implemented in future incarnations of the course.

Table 1: Outline and timetable of topics discussed in the course.

Week 1:	Introduction to Inferno; abstractions and names
Week 2:	Overview of the Limbo programming language
Week 3:	Data types in Limbo and the Dis virtual machine
Week 4:	Inferno kernel overview
Week 5:	Inferno kernel device drivers
Week 6:	*Break : no class*
Week 7:	C applications as resource servers: built-in modules and device drivers
Week 8:	Case study
Week 9:	Platform independent interfaces: Limbo GUIs; project update
Week 10:	Programming with threads, CSP
Week 11:	Debugging concurrent programs; Promela and SPIN
Week 12:	Factotum, Secstore and Inferno's security architecture

2. Course Outline

The original goal in designing the course, had been to focus on the topic of *concurrency*, from the perspectives of both theory and practice. It was intended to explore models of computation such as CSP [4], CCS [5], and the π-calculus in the context of the Limbo programming language, and to investigate the behavioral validation and verification of concurrent Limbo programs using the SPIN model checker [6].

Based on the enrollment of the course, and the interests of the students, the course evolved to include more discussion of the implementation of Inferno, and the implementation of the emulator and native kernel. The course duration was one semester (12 weeks), with two hour-long lectures a week. In all, 15 prepared lectures were spread across these 24 meeting times. An outline of the topics discussed is listed in Table 1.

An overview of the 15 core lectures is presented next, along with some of the actual questions posed by students during or after the lectures. In many cases, the questions have been reworded to include sufficient context.

2.1. Lecture 1

Course overview; syllabus; introduction to Inferno; demo of native Inferno (in VirtualPC) and emulator; demo of sample applications (Charon, shell, games). **Questions:** "How many people use Inferno / how large is the developer community?", "What applications is it good for?".

The usually common question of "How is Inferno different from Java?" was not raised since by the end of the first lecture, the structure of the Inferno *system* (language atop virtual machine, atop host OS or native kernel, atop hardware) had been described and distinctions from, and analogies to other systems drawn.

2.2. Lecture 2

The concept of abstracting system resources; names (files) as an abstraction for system resources; structure of the native and emulated Inferno systems (Limbo threads, Dis VM, Styx, device driver interface, etc.); per-process name spaces; discussion of Unix `/dev` and `/proc`, `ioctl`, `mknod` etc.; unification of both resource access and control in Inferno device driver filesystem interface; the Mount device and in-kernel `Chan` structures; overview of Styx and snooping on Styx messages (with demo); demo of the C-language Styx server from the Inferno distribution to illustrate relation between filesystem operations and generated Styx messages. **Questions:** "Is this similar to Unix `/proc`?", "Is this similar to FUSE?", "is Styx similar to NFS/AFS?", "Are Limbo channels related to Plan 9's plumber?".

The lecture began with a stress on referring to the entries in the "filesystem" as *names*, rather than

as *files*, and a stress on the fact that these names are just an abstraction to system resources and are not necessarily files in the traditional sense of disk-based data. This prevented the oft-heard question of "If all resources are represented as files, won't there be a lot of disk accesses?". Those students who recognized the similarity with the Unix `/dev` and `/proc` filesystems only realized the real difference when the question of the role of `ioctl`'s in Unix was broached. The examples of remote debugging via a mounted `/prog`, and the immediate visibility of remote processes in the output of `ps` after the mount helped solidify understanding.

2.3. Lecture 3

Introduction to Limbo; a "Hello World" program in Limbo; Limbo module interface and module implementation; compiled Limbo programs — what goes into a `.dis` binary; comparison via a demo, of the `disdump` of the Hello World binary, and the `objdump -d` of a compiled C language Hello World; Limbo data types, modules, dynamic loading of modules; Limbo language genealogy and related or similar programming languages. **Questions:** "Can you define new types in Limbo?"; "What is the reason for duplicating the definition of the `init` function both in the module interface definition and in the body of the Limbo program?" (This question is really referring to the function type definition that appears in the module interface type, and the function implementation that appears in the body of the Limbo program); "Are there characters as a basic type?", "If you were in the shell or Acme and wanted a literal representation of a Unicode character, is there a magic keystroke or escape sequence?".

Most of the students in the class were undergraduates from the computer science program, and had been exposed to ML. Thus many students drew similarities between Limbo and ML.

2.4. Lecture 4

Limbo data types; demo implementing several example Limbo programs; introduction to Limbo channels; large example: the prime number sieve ("streams" implementation). **Questions:** "Are there functions that are generics that will take a list of anything?", "Can you do pattern matching on tuples?", "How do you define a type?".

Although not asked in this particular course, from experience, a common class of questions posed by people first hearing about Limbo is the "Does it have *X*?", where *X* is a buzzword or a topic dear to the questioner's heart, and has on occasion been many things ranging from "classes and inheritance" or "introspection" to "reflection" and many things in between.

2.5. Lecture 5

Limbo data types; Unicode and ASCII; the UTF-8 multi-byte encoding for Unicode; strings, lists, arrays, slices, tuples, ADTs; discussion of course project ideas. **Questions:** "Is string indexing an index into the characters in the string (type `int`), or into the bytes in the UTF-8 representation of the vector of characters that make up the string?"; "If you index at an arbitrarily far point beyond the end of a string, will the runtime automatically reallocate space for a longer string?"

2.6. Lecture 6

ADTs; Dis VM — machine model, brief look at Dis VM specification document. **Questions:** "Can you assign to a function member within an ADT, since the syntax seems to indicate that you can?"

2.7. Lecture 7

Inferno kernel and emulator overview; source tree layout; overview of source files making up emulator implementation; build tools; build configuration files. **Questions:** "Can the emulator be compiled with GCC?"; "Can the emulator be compiled under Cygwin on Windows?"; "Can the native kernel be compiled with GCC?"; "Why can't the native kernel be compiled with GCC?".

2.8. Lecture 8

Inferno emulator structure; emulator source files — `dat.h`, `fns.h`, `error.h`; the `Chan`, `Dev`, `Dirtab`, `Proc` and `Osenv` structures; compiling the emulator.

While there were no questions for this material, that is not necessarily an indicator of the material being intuitively obvious, or of clarity of the exposition.

2.9. Lecture 9

Native kernel overview; kernel source; supported architectures; `dat.h`, `portdat.h`; the `Chan`, `Dev` and `Proc` structures; compiling a native kernel; the Plan 9 C compilers; deviations from ANSI C in the native kernel implementation; kernel build configuration file; compiling a native kernel. **Questions:** "Does the Plan 9 C compiler generate platform-retargetable object code?"; "What are the details of the hardware platforms supported, e.g. `ipengine`, `js`?".

A topic raised in the discussions of this lecture, was the desire to get Inferno running on some new platform *du jour*. As is often the case with such desires, the motivation for getting Inferno running on the new target, other than its educational value, was debatable. The discussions however offered an opportunity to examine what applications would be best implemented using Inferno, and whether a native kernel deployment or the emulator running over a host operating system would be best.

2.10. Lecture 10

Native kernel initialization; `l.s`, `main.c`, `machinit()`, `archreset()`, `confinit()`, `links()`, `xinit()`, `poolinit()`, `poolsizeinit()`, `trapinit()`, `clockinit()`, `procinit()`, `chandevreset()`, `userinit()`, `chandevinit()`, `schedinit()`.

The manner in which the material in this lecture was presented, as a walk-through of the native kernel initialization, resulted in few questions or discussions. In retrospect, preceding it with a longer discussion of the initialization process, before looking at source code, might have yielded a more lively lecture.

2.11. Lecture 11

C language applications as Inferno resource servers — device drivers, built-in modules and Styx servers; the `Math` built-in module; implementing built-in modules; `math.m`; generating C stubs using the Limbo compiler; built-in module initialization, `libinterp`. **Questions:** "When would you want to use a built-in module versus a device driver?"; "Is it safe to implement new functionality as built-in modules?"; "Do built-in modules run in 'ring-0' (i.e., at the highest privilege level on the x86 architecture)?".

A cause of great concern among students was the fact that built-in modules were introduced as a way to embed functionality implemented in C within the emulator or native kernel, but that there was little protection against such code wrecking havoc to the entire system. This led to a discussion of the preference of implementing such functionality as a Styx server external to the emulator or native kernel, e.g., using `libstyx`.

2.12. Lecture 12

The Mount driver, cycles in mount points.

2.13. Lecture 13

Hardware parallelism; CSP; CSP historical perspective and context (Dijkstra's guarded commands, coroutines, Algol 60, Pascal); CSP "commands", processes and parallel composition; channels; alternation and repetitive commands; structure matching in CSP; coroutines, subroutines and monitors in CSP; CSP examples.

2.14. Lecture 14

More CSP examples; relation between Limbo and CSP constructs (Limbo `case` and `alt` compared to CSP's '□' operator.

2.15. Lecture 15

CSP review; concurrency in applications — concurrent applications can be tricky to "get right"; specification of concurrent behavior; Promela and SPIN; examples; using SPIN; message sequence charts in SPIN; SPIN demo.

3. Summary

This paper outlined the content of a semester-long course on the Inferno operating system and the Limbo programming language, and the questions and discussions that arose from the course. The course material, consisting of a total of approximately 400 presentation slides, and incorporating digital photographs of the blackboard from in-class discussions and illustrations, is available to the general public.

Many of the concepts in the Inferno and Plan 9 operating systems are outside the common knowledge of many students and practitioners in the computing sciences, and it was the objective of this paper to provide a detailed description of one approach to explaining these concepts, and to detail the responses and questions arising from the presentation thereof.

References

[1] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: the story of Squeak, a practical Smalltalk written in itself. ACM SIGPLAN Notices **32**(10) (1997) 318–326

[2] Wirth, N., Gutknecht, J.: The Oberon system. Software-Practice and Experience **19**(9) (1989) 853–893

[3] The Institute of Electrical and Electronics Engineers (IEEE): Chapter 16: Thread Management. In: Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]. (1996)

[4] Hoare, C.: Communicating sequential processes. Comm. ACM **21**(8) (1978) 666–677

[5] Milner, R.: A calculus on communicating systems. Lecture Notes in Computer Science **92** (1980)

[6] Holzmann, G.J.: The model checker spin. IEEE Transactions on Software Engineering **23**(5) (1997) 279–295