Fall-12: Introducing the basics of user-level multithreading

Ana Lúcia de Moura, Noemi Rodriguez

Departamento de Informática
Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro, Brazil
amoura, noemi@inf.puc-rio.br

***Abstract*—Current parallel libraries and languages often rely on user-level threads to allow for massive concurrency. However, even junior or senior students often have difficulties with the understanding of the pros and cons of user-level threading, due to a lack of concrete experience with its implementation. In this poster, we describe our effort to introduce this concept in the introductory Computer Systems course at PUC-Rio. This is a required course for three of PUC-Rio's undergraduate programs: Computer Engineering, Computer Science and Information Systems. During the course, students are exposed to basic concepts underlying computer systems, learning and exploring the ways in which these concepts provide support for implementing several abstractions present in typical imperative programming languages. Because the course explores extensively the concepts of execution stack and global data area, it is an ideal point in the curriculum to introduce user-level threads. We describe the theoretical and laboratory material we developed on the topic, in which we explored the concept of coroutines and a basic library that can be built on to implement symmetric and asymmetric coroutines.**

***Keywords*-user-level multitasking, coroutines, computer systems**

I. Introduction

One of the important concepts in the NSF/TCPP proposed curriculum is the idea that a better understanding of parallelism and distribution will not be achieved solely by the introduction of new courses, but may instead require remodelling of many traditional courses. Some concepts need to be introduced in a cross-cutting manner, either because they are needed in many areas and cannot be handled as a separate issue, as is the case with fault tolerance, or (not exclusively) because they rely on concepts from different areas. In teaching concurrent programming over the last years, we have felt that one concept in this second category is the basic notion of multithreading itself. The tradeoffs and pitfalls involved in multithreading may involve discussions in the different areas of computer architecture, computer systems, compiler construction, and programming languages [1], [2]. We have felt that students have specific difficulties grappling with the differences between system and user-level multithreading. It is usually difficult for them to interpret pros and cons and to understand why many currently popular systems and languages rely on user-level multitasking to attain massive concurrency.

Thinking about this, we came to the conclusion that the idea of user-level multitasking is described in a much too abstract form to students who do not have sufficient maturity in the related areas. Students usually have hands-on experience with systems-level threads (or with processes) in a conventional operating systems course, in which course projects typically involve tinkering with the scheduler. On the other hand, they typically have no experience with user-level threads, and often show some wonder at the possibility of implementing a scheduler inside a regular program.

At PUC-Rio, we teach an introductory Computer Systems course with is required for all undergraduate programs in computer science. Because of the attention given in this course to the implementation of the execution stack, we decided it would be an ideal place for students to have a first experience with multiple user-level stacks and thus a concrete understanding of user-level threads. In this poster, we report on how we introduced lecture and lab material on coroutines in the Computer Systems course in 2012, with special emphasis on the learning materials used in the lab.

II. Context

At PUC-Rio, students in the three undergraduate programs related to Computer Science (Computer Engineering, Computer Science, and Information systems) take a course on Systems Software, typically around their fourth semester. This course has much in common with conventional Computer Systems courses but it has a strong emphasis on the software perspective, and relies heavily on selected material from the book by Bryant and O'Hallaron [3]. During this course, students are exposed to basic concepts underlying computer systems, learning and exploring the ways in which these concepts are used in the implementation of typical data and control abstractions. One of the main topics is the stack-based implementation of subroutines.

Our Systems Software course uses the IA-32 architecture as its main example and is by this time well organized into a collection of one-week modules, each of them containing a set of concepts that must be covered by the instructor and a hands-on session (see

http://www.inf.puc-rio.br/~inf1018). We begin by covering computer arithmetic and the representation of different data types provided by the C programming language (unsigned and signed integers, floating point numbers, arrays and data structures). After that, we introduce the IA-32 assembly language, and explore the "translation" of C basic statements and control structures (assignments, conditionals, loops) to their machine-language counterparts. We next cover in depth the implementation of subroutines, including parameter passing and activation records. Stack allocation of local variables is motivated by examples using local arrays or containing function calls. Next, there is a section on interrupts and traps, in which the students learn how the operating system is activated and how interrupt handlers work. The final module of the course covers the linking process.

A. Changes to the syllabus

Our goal was to allow students to gain an understanding of how concurrency can be implemented at application level. We introduced an extra (class, laboratory) pair, presented to the students after they had assimilated the concepts related to subroutines. In the new topic, students learned about coroutines and their typical uses. For the theoretical presentation, we used some of the material from Michael Scott's book on Programming Languages [4]. As an example of a coroutine library provided by a programming language, we discussed Lua coroutines [5] and showed some typical uses of coroutines implemented in Lua. The use of examples in Lua allowed us to motivate coroutines concentrating on the their logic and not in coroutine details.

We also developed a small coroutine library (http://www.inf.puc-rio.br/~inf1018/coro.tar) and devoted one hour in class for navigating with the students through its code. A core module (`core.s`) is the only part that needs to be written in assembly, and provides a very basic `icoro_transfer` that is responsible for changing the value of the program counter and of the stack pointer while saving the old values. This core code is used in two different coroutine implementations, providing symmetric and asymmetric coroutine variants [6]. The provision of these two interfaces was intended to facilitate the discussion of different uses of coroutines, such as iterators and cooperative multitasking. We then devoted a two-hour class to a lab where the students used this library to implement a very simple scheduler (http://www.inf.puc-rio.br/~inf1018/corrente/labs/lab16.html). Unfortunately, the course material is at this time available only in Portuguese.

B. Evaluation

Because we have not yet formally introduced coroutines into the course syllabus, this topic was not included in the course exams. However, because the two authors personally assist students during lab sessions (there are no teaching assistants), we could assess students' understanding of the topic.

In the lab, students implemented a scheduler for an application-level cooperative multitasking application over the provided library. With this exercise, students could acquire a concrete understanding of multitasking and we had an opportunity to review and discuss the differences between system and application-level multitasking, and their respective pros and cons.

We felt students were quite motivated by this new topic, both in classroom and lab. In the classroom, we felt it was a good choice to use Lua in the examples: the mention of example uses of coroutines in game programming was specially motivating. In the lab, students were pleasantly surprised with their understanding of the internal working of a scheduler. We would like to extend this topic so as to have students program the coroutine library itself. To this end, we would have to extend the time dedicated to the topic to two weeks. In the first week, we would discuss the basic concepts and present the core library with only the asymmetric variant, and repeat the scheduler assignment in the lab. This would allow students to develop familiarity with the concept of transferring control before being presented to different variants of coroutines. In the second week, we would have students themselves develop the symmetric variant of the library. This would allow an even more concrete understanding of the stack manipulation needed to implement user-level multitasking.

Another direction for future work is the introduction of related topics in further courses. One of the ideas that we are interested in is introducing changes in the Operating Systems course to explore the ideas behind alternative implementations for thread pools. But these ideas are still under initial discussion.

REFERENCES

[1] S. V. Adve and H.-J. Boehm, "Memory models: a case for rethinking parallel languages and hardware," *Communications of the ACM*, vol. 53, no. 8, pp. 90–101, 2010.

[2] H.-J. Boehm and S. V. Adve, "You don't know jack about shared variables or memory models," *Communications of the ACM*, vol. 55, no. 2, pp. 48–54, 2012.

[3] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 2nd ed. Prentice-Hall, 2011.

[4] M. Scott, *Programming Language Pragmatics*, 3rd ed. Morgan Kaufmann, 2009.

[5] R. Ierusalimschy, *Programming in Lua, Second Edition*. Lua.Org, 2006.

[6] A. L. Moura and R. Ierusalimschy, "Revisiting coroutines," *ACM Transactions on Programmming Languages and Systems*, vol. 31, no. 2, pp. 6:1–6:31, Feb. 2009.