Interactive Labs for Experimental Systems in Education

Executive Summary

Fabián E. Bustamante
EECS, Northwestern University
Evanston, IL, USA
fabianb@cs.northwestern.edu

1. MOTIVATION

Experimental systems combines theoretical science with the creativity and skills needed to translate good ideas into useful systems that work in realistic settings. This approach is central to the research process, both as the ultimate validation for new theories and methods, and as a way of exploring tunable parameters and understanding new emergent behavior. I believe this experimental approach is essential to an undergraduate education, and may help reverse the declining skillsets and competitiveness of U.S. computer science and engineering undergraduates.

Adopting an experimental approach in education is a challenging task for instructors as it requires us to integrate and balance the theoretical foundations and practical aspects of the topic at hand in a relatively short period of time. In operating systems, for example, the instructor is normally left with two unsatisfactory options: either to superficially focus on one small aspect of a complex but real OS, or to target a more thorough understanding of a "toy" one. The complexity of the systems under study is partially to blame. They are large, involved beasts of thousands to millions of lines of code, implementing various, tightly interconnected, subsystems.[1] A single academic term is simply not enough to understand, let alone modify, such complex systems in any meaningful way. The task is not made simpler by highly heterogeneous student pools that may comprise a wide range of skills, maturity levels and motivations.

Many instructors seem to favored the second, toy-based, model for introductory operating system courses. Following this model, 90% of the term is dedicated to the classical topics of processes, memory, storage and protection and, if time permits, one or two lectures browse on security and distributed systems topics. Typical projects in these courses range from writing a tiny shell or adding a simple system call to an existing OS, to comparing kernel memory allocators or implementing a small file system on a virtualized disk. By the end of the term most students have a good understanding of key concepts such as processes and threads, synchronization and virtual memory. Challenged, however, with a more systemic question that looks at the odd interactions between those concepts, and the majority of them will be at a loss.

[1]Windows XP is over 40 millions source lines of code (SLOC), while RedHat Linux 7.1 and Sun Solaris are about 30 million and 10 million SLOC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

2. AN INTERACTIVE LAB MODEL

I argue that a promising approach to this challenge is to adopt a lab-like, hands-on interactive environment not unlike those in physical-chemistry courses. Armed with a good understanding of a system's components and provided with a carefully crafted set of exercises built on a tool for dynamic system instrumentation, students will be better able to create useful mental models of the complex systems interactions – a level of understanding virtually impossible to attain in most traditional, term-long courses. Such a laboratory could also serve as a perfect environment where to introduce students to rigorous experimental methods and emphasize the experimental origins of scientific knowledge and development.

2.1 Dynamic Instrumentation as a Building Block

Students in these labs will experiment with systemic behavior in real operating systems using a set of exercises built on a tool for dynamic system instrumentation.

There has been a large body of work on dynamic system instrumentation. From Miller et al.'s Paradyn tool [3, 4] to more recent work on Linux' DProbe [5]. Dynamic instrumentation has proved to be useful for debugging [1] and performance optimization [2]. However, until the relatively recent release of Solaris 10, most tools were poorly supported and/or part of research focused projects.

DTrace [2] is a new tool for instrumentation of production system, made publicly available as part of Solaris 10. DTrace allows for dynamic instrumentation of both user- and kernel-level software, in a unified and safe fashion. By relying only dynamic instrumentation, DTrace has zero probe effect when not explicitly enabled. The tool includes a C-like high-level control language, *D*, that allows users to interactively define predicates and actions associated with any given point of instrumentation. D allows access to the kernel's native types and global variables, includes support for all ANSI C operators, and offers several kind of user-defined variables such as thread-local variables and associative arrays. D programs have a very simple structure that resem-

```
syscall::read:entry
{
    self->t = timestamp;
}

syscall::read:return
/self->t/
{
    printf(``%d/%d spent %d nsecs in read\n'',
           pid, tid, timestamp - self->t);
}
```

Figure 1: A simple DTrace script example from [2]. The script outputs the amount of time a thread spends in a `read(2)` system call.

bles `awk(1)`. Each program consists of one or more clauses that describe the instrumentation to be enabled by DTrace. Each clause has the form:

```
probe-descriptions
/predicate/
{
   action-statements
}
```

Where *probe-descriptions* specify the set of probes to instrument (e.g. `syscall::entry`, all entries to a syscall), *action-statements* is a list of statements that could be as simple as a `printf`, while */predicate/* acts as a filter to limit the execution of action statements (e.g. /execname = "foo"/ will restrict the action to those `syscall::entry` for an executable named "foo"). Figure 1 shows a simple example of a script for computing the time a thread spends in a `read(2)` system call.

2.2 Interactive Labs in Operating Systems

Clearly the proposed idea of interactive labs for understanding the complex interactions among the components of a large system could be applied to variety of courses, from architectural design to distributed systems. Still, I believe that its use in operating systems has the potential for wider impact within our field. Operating systems is, after all, a commonly required course and one of the toughest classes in most programs. Northwestern is not exception. Operating systems (EECS 343) is currently a required course for Computer Engineering and a "strongly" recommended course for Computer Science. It is also one of those courses students forums advise to take only in a very lightly loaded term. Students come to it with some understanding of computer architectures, a bit of assembly languages, and different levels of C programming experience. They are expected to leave from it with not only a high-level understanding of operating systems concepts and common algorithms, but also with a solid grasp of the OS components' complex interactions and quite a bit of system-level programming experience. All in the term of 10 weeks.

I am currently exploring with different models for the inclusion of labs in an existing course. A fairly straightforward alternative is to include them as part of a capstone project based, for example, on the performance debugging of a systemic problem.[2] Another, more involved model, could rely on labs as the discussion starter for each of the core operating systems topics.

[2]Cantrill et al. [2] offer an interesting experience report in the context of a production SunRay server.

3. CONCLUSIONS

Adopting an experimental approach in education is a challenging task for instructors as it requires us to integrate and balance the theoretical foundations and practical aspects of the topic at hand in a relatively short period of time. I argue that a promising approach to this challenge is to adopt a lab-like interactive environment similar to those found in more traditional science courses. Armed with a good understanding of a system's components and provided with a carefully crafted set of exercises built on a dynamic instrumentation tool, students will be better able to create useful mental models of complex systems interactions – a level of understanding virtually impossible to attain in most traditional, term-long courses.

4. REFERENCES

[1] Auguston, M., Jeffery, C., and Underwood, S. A monitoring language for run time and post-mortem behavior analysis and visualization. In *In Proc. International Workshop on Automated and Algortihmic Debugging* (Ghent, Belgium, 2003).

[2] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. Dynamic instrumentation of production systems. In *Proc. of USENIX ATC* (Boston, MA, June 2004).

[3] Hollingsworth, J. K., Miller, B. P., an Oscar Haim, M. J. R. C., Xu, Z., and Zheng, L. MDL: a language and compiler for dynamic program instrumentation. In *In Proc. of the International Conference onParallel Architectures and Compilation Techniques* (1997).

[4] Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T. The Paradyn parallel performance measurement tool. *IEEE Computer 28*, 11 (1995), 37–46.

[5] Moore, R. J. A universal dynamic trace for Linux and other operating systems. In *Proc. of USENIX ATC* (Boston, MA, June 2001). FREENIX Track.