Teaching Operating Systems: Just enough abstraction

Philip Machanick

Department of Computer Science
Rhodes University
`p.machanick@ru.ac.za`

Abstract. There are two major approaches to teaching operating systems: conceptual and detailed. I explore the middle ground with an approach designed to equip students with the tools to explore detail later as the need arises, without requiring the time and grasp of detail needed to understand a full OS implementation. To meet those goals, I apply various strategies to different concepts, for example, faking the detail and using techniques from computer architecture simulation. The course aims to give students a better sense of how things work than a conceptual approach without the time required for a full implementation-based course.

1 Introduction

Operating systems (OS) courses divide roughly into a general survey of OS features and variations – a conceptual approach – and those that dive into detail. The latter category includes use of a real OS (usually these days one with free source such as the Linux kernel), or an OS designed specifically for teaching and research such as MINIX [23].

At Rhodes University (South Africa's smallest research-intensive university), the third year OS module in a Computer Science major takes 4 weeks each with 5 lectures and one practical (of 3 hours), and students usually take two third-year subjects, allowing 20 hours per subject per week. Timetabled contact time of 8 hours leaves 12 hours a week for independent work. That is insufficient time to go into much detail of a full-scale or even cut-back OS such as MINIX.

Since most programming courses at Rhodes are taught using higher-level languages that manage memory and abstract away all the details of the machine, one of my goals in the OS module is to reinforce exposure to the machine layer and notions like machine addresses, only seen in a small part of our curriculum. Finally, learning really requires some exposure to how professionals in the real world work [11] – so some aspect of developing code typical of OS internals is necessary for a real understanding of the area.

The approach I describe here attempts to achieve some of the benefits of a full-scale implementation-oriented OS course without the time commitment required to do so. Strategies used include:

SACLA 2016 **preprint** *to appear in CCIS 642, Springer-Verlag*

check http://homes.cs.ru.ac.za/philip/Publications/_SACLA/OSjustEnoughAbstr-2016-abst.html
for published version

– *faking part of the system* – e.g, to illustrate file system variations like a file allocation table (FAT) [5] or inodes [13], I implement RAM-based structures illustrating how the disk-based pointers would be organized
– *architecture simulation techniques* – trace-driven simulation [24] allows a small part of a system to be simulated provided a trace of memory accesses is available to drive the simulation

In the remainder of this paper, I provide background to relevant educational theory and other approaches to teaching operating systems. I go on to explain the design of my course followed by more detail. I share experience from running the course this way and wrap up with conclusions.

2 Background

Earlier theories of learning focused on cognition. The constructivist model, for example, inspired by the work of Piaget [18], was based on different levels of sophistication of building mental models [3], similarly to Bloom's Taxonomy, which ranks different levels of problem solving in terms of sophistication [10]. Social constructivism adds to constructivism the notion that there is a social aspect in learning – that construction of knowledge, while a cognitive process, is influenced by interactions with others [7]. The social construction model goes a step further and divorces learning from cognitive models, focusing instead on how knowledge is created by social interaction [11,4].

Whether we accept a cognitive view of learning or change our focus to a purely social model, the consensus is that learning requires *doing* – a strong argument against a pure survey approach. Understanding an OS requires overcoming a number of misconceptions [16]; it is hard to see how such misconceptions can be overcome without a strongly practical component to an OS course.

In the early 2000s, *instructional operating systems* such as Nachos, Topsy and Yalnix were designed to abstract key concepts to simplify teaching [2]; Pintos is more recent [17]. Some teach the Linux kernel [15,8,6] and others Windows internals [21]. All these whole-system approaches require significant time to learn the basics before getting into detail.

3 Course Design

For the Rhodes OS course, the approach I take is to cover major ideas in lectures and drill down to implementation in practicals. I provide detailed notes [12] and work through examples and concepts in class, interspersed with C programming techniques with the aim of preparing the class for the next practical exercise.

Main headings follow a typical OS course outline:

– *The Kernel*
  - system calls and interprocess communication (IPC)
  - what goes in kernel *vs.* user space – microkernels *vs.* monolithic kernels

    - what the kernel does
- *Schedulers*
    - theoretical approaches
    - practical approaches
    - examples: Windows and Linux schedulers
- *IO and Files* – including inodes and FAT file organiation
    - device interface
    - files and devices
    - performance – including speed as well as reliability and fault tolerance
    - protection and security
    - other device types
- *Memory* – mostly virtual memory (VM) using pages
    - history and rationale for memory management
    - key concepts of VM
    - more advanced concepts
    - examples including real machines and translation lookaside buffers (TLBs)
- *Parallel Programming* – including Pthreads, UNIX-style processes and IPC
    - concepts
    - launching
    - sharing and communication
    - synchronization
    - distributed systems and the cloud
    - parallel programming hazards

All of this can relatively easily be covered with a survey approach; there are good OS texts that do just that [23,22]. The challenge is how to approach these topics in more depth without a full implementation of an OS – or more specifically, in a relatively short time.

The approach I take is to implement small fragments of an OS that can be designed, implemented and tested independently so that a whole OS does not need to exist or be understood to do practical work. I describe here two major approaches: implementing a small, simplified subset of functionality in a way that can be tested in isolation and using trace-driven simulation to implement functionality that would normally be driven by execution of user-level code.

I also illustrate user-level functionality by showing how to use system calls and standard libraries that implement functionality that illustrates core concepts like synchronization and parallel programming.

3.1 Small Subset

File system concepts can be implemented at least to some extent without the whole OS. The key concepts I want to illustrate in the course are the way the file system can be layered (as in a UNIX-style file system with a virtual file system on top of which the actual file system is implemented) and the pointer structure of an inode or FAT file system.

```
typedef struct FS_attributes {
   char           fstype[FSTYPEN];  // type of FS
   blocksize_t    blocksize;
   blockpointer_t numblocks;
   blockpointer_t maxfiles;
   blockpointer_t bitmapSize;       // in blocks
   blockpointer_t directory;        // must be followed by first_fileptr
   blockpointer_t first_fileptr;    // must be followed by freespacelist
   blockpointer_t freespacelist;    // must be followed by first_data_block
   blockpointer_t first_data_block;
   blockpointer_t mappedblocks;    // minus attributes, directories, etc.
} FS_attributes;
```

Fig. 1. Highly simplified VFS structure. It includes just enough detail to find blocks that are either system overheads such as directories or file blocks.

3.2 Trace-Driven Simulation

To implement trace-driven simulation, I generate traces using Pin [9], which I use to generate a trace file out of a user-level executable containing a record of instructions fetched (as their address) and addresses read and written. To approximate the effect of interrupts, I add into the trace files artificially-generated records of interrupts, each with the latency of handling the interrupt.

3.3 User-Level Examples

Synchronization, process launching and threads, while good to understand at the kernel level, are hard enough at the user level that I consider it adequate to use user-level coding for these examples. Areas covered include Pthreads [14], UNIX-style `fork` and various modes of IPC (shared memory, memory maps and pipes). I also review various synchronization primitives including mutexes, spinlocks and barriers – including efficiency and implementation issues. The class does practical work to implement examples that are designed to illuminate principles.

4 Course Detail

To illustrate how all this works in practice, I provide examples of practical problems set, covering the various techniques. For a simplified subset that can be tested in isolation, I use the example of implementation of a file system. For trace-driven simulation, I use two examples: scheduling and VM. Finally, I illustrate the use of user-level examples with parallel programming.

4.1 Small Subset: File System

To illustrate how a file system is implemented, I provide code that crudely approximates to the split between a virtual and actual file system. A virtual file

```
struct Inode {
   // attributes: permissions and path
   char path[NAMELENGTH]; // byte 0 nonzero if a valid inode
   unsigned int permissions;
   unsigned int size;
   FS_t *filesystem;
   blocksize_t blocksize; // property of file system but fixed once set
   blockpointer_t direct_pointers[NUMBERDIRECT]; // size must be constant
   blockpointer_t single_indirect_pointers; // points to FS pointer block
};
```

Fig. 2. Simplified inode. I omit many details (e.g., timestamps, link count).

system (VFS) was originally designed to hide implementation details such as whether the file system is local or remote [20]; in my approximation to this, a low-level file system implements block operations on a device simulated in RAM that can be used without needing to know where blocks are stored, capturing the essence of a VFS without the complexity. This simplified VFS (Fig. 1) allows implementing operations on an inode-based system to create, remove or extend a file – or doing the same using FAT. A bitmap representing free or allocated blocks provides exercises in bitwise operations. Conceptual challenges students must deal with include understanding that file system pointers are not the same as memory pointers (they refer to device blocks, not bytes in main memory) and that data structures used to represent files can be complex to navigate.

Figure 2 illustrates my minimalist inode structure. It contains a pointer to a data structure defining the VFS in which it is contained; all other "pointers" are disk block numbers, as determined by the VFS. The VFS knows that a file system contains certain overheads – directories, top-level file pointers – and the actual file system initializes it with sizes of these overheads.

4.2 Trace-Driven Simulation: scheduling and virtual memory

Pin allows me to produce trace files that mark memory addresses as one of read ("`R`"), write ("`W`") or instruction fetch ("`I`"). I add in fake interrupts at regular intervals, each of fixed latency ("`X`"; the number in the file in this case is the latency, not an address). Here is an example of an extract from a trace file:

```
I 0xb78882a0
W 0xbfd913d4
X 0x3E8
R 0xbfd91564
```

In this example, there is an interrupt with latency (in clock ticks) `0x3E8` = 1000_{10}. Each instruction fetch is assumed to add 1 clock tick. If I am not simulating memory hierarchy, reads and writes are fully pipelined (add no latency).

To create a workload, my simulator reads in a list of trace file names that represent a process per trace file.

Scheduling To keep things simple I assume that all interrupts are only processed once a waiting process reaches the head of a single wait queue. To simulate scheduling, it is only necessary to process instruction fetches and interrupts from the trace file; memory reads and writes are ignored. If a process is interrupted, it goes to the wait queue until it reaches the head of the wait queue and after than becomes ready only after its latency has expired. This framework allows comparison of variations, e.g., round-robin scheduling and multilevel feedback queues (as in Windows [19] and some versions of Linux [1]).

While avoiding the true complexity of a scheduler, in the spirit of "just enough abstraction", students see the main issues.

Virtual Memory VM is even harder to code at the true hardware level than scheduling, since implementation has to match hardware functionality closely. Trace-driven simulation simplifies exploring variations like alternative page table structures and the functioning of a TLB. By including reasonable numbers for latency of operations, even if the detail is not fully simulated, it is possible to illustrate the performance impact of design choices. In addition, giving the students an example and asking them to implement a variation makes it possible for them to get a sense of how a real system is implemented.

Given a single-level page table, implementing a two-level page table provides a reasonably challenging programming example. Another example of similar levels of difficulty and insights is evaluating the effect of a TLB.

4.3 User-Level: parallel programming

Finally, to illustrate concepts related to processes, threads and IPC, user-level programming can provide good insights. Examples I use include:

- *threads* vs. *processes* – given an example of one, recode using the other
- *shared memory* vs. *memory maps* – again, recode in the other type
- *synchronization* – focus on a subset of types of options (barrier, mutex, etc.)
- *IPC primitives* – coding using pipes adds another dimension

To fit the limited time, I vary what is covered in lectures *vs.* in practicals.

5 Experience

My experience of explaining concepts like multilevel page tables and TLBs in lectures is that they are very difficult concepts to grasp in the abstract. Parallel programming is another area where doing is really required to learn. Some areas like scheduling are easier to learn conceptually, though conceptual texts present scheduling in a theoretical way unrelated to real OS design [22]. A case study of Linux scheduler evolution is more interesting and also exposes students to the debate about free versus proprietary software (why did Linux evolve so fast, while the Windows scheduler has not changed much in overall design since Windows

NT?). It is difficult to make this sort of debate come to life without the students having a feel for how things are actually implemented.

That students battle with low-level concepts like pointers is not a reason to avoid them. If they must learn them somewhere, an OS course – at the interface between hardware and software – is a logical place to introduce them. An OS course also illustrates how pointers can differ in different layers of the system (file system pointers refer to disk blocks not bytes in memory).

6 Conclusion

The real test of any course is whether it helps the students grow – and that can be hard to measure in the short term particularly with a final-year course. The class generally finds the course challenging, as we move rapidly to new concepts and they are drawing on a very limited prior exposure to low-level coding in C (one 3-week module in second year). However it would be a lot more challenging were the course to be based on a real fully-implemented OS.

Students who have taken the course and return after a few years with reports on its usefulness will be the real test of the value of the approach; the course has not been running long enough in its current form for such an evaluation. My own experience is that students taught using this just enough abstraction approach have a better appreciation of implementation and design issues than those taught using a purely theoretical approach.

As the course evolves, I plan on varying the detail – changing for example where I use the three strategies (small subset, trace-driven simulations) and user-level coding – to find the right mix. In the meantime I invite others grappling with finding the right balance between abstraction and detail to share ideas.

References

1. Aas, J.: Understanding the Linux 2.6. 8.1 CPU scheduler. Tech. rep., Silicon Graphics, Inc. (2005), `http://joshaas.net/linux/linux_cpu_scheduler.pdf`
2. Anderson, C.L., Nguyen, M.: A survey of contemporary instructional operating systems for use in undergraduate courses. J. Comput. Sci. Coll. 21(1), 183–190 (Oct 2005), `http://dl.acm.org/citation.cfm?id=1088791.1088822`
3. Ben-Ari, M.: Constructivism in computer science education. In: Proc. 29th SIGCSE Tech. Symp. on Computer Science Education. pp. 257–261. SIGCSE '98, ACM, New York, NY, USA (1998)
4. Bijker, W.E., Hughes, T.P., Pinch, T., Douglas, D.G.: The social construction of technological systems: New directions in the sociology and history of technology. MIT press (2012)
5. Chen, J.B., Endo, Y., Chan, K., Mazières, D., Dias, A., Seltzer, M., Smith, M.D.: The measured performance of personal computer operating systems. ACM Trans. Comput. Syst. 14(1), 3–40 (Feb 1996)
6. Dall, C., Nieh, J.: Teaching operating systems using code review. In: Proc. 45th ACM Tech. Symp. on Computer Science Education. pp. 549–554. SIGCSE '14, ACM (2014)

7. Kim, B.: Social constructivism. Emerging perspectives on learning, teaching, and technology 1(1), 16 (2001)
8. Laadan, O., Nieh, J., Viennot, N.: Structured Linux kernel projects for teaching operating systems concepts. In: Proceedings of the 42nd ACM Tech. Symp. on Computer Science Education. pp. 287–292. SIGCSE '11 (2011)
9. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proc. 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 190–200. PLDI '05 (2005)
10. Machanick, P.: Experience of applying Bloom's Taxonomy in three courses. In: Proc. Southern African Computer Lecturers' Association Conf. pp. 135–144 (2000)
11. Machanick, P.: A social construction approach to computer science education. Computer Science Education 17(1), 1–20 (2007)
12. Machanick, P.: 2OS: more programming from the machine up. Rhodes University, Grahamstown (2016), `http://homes.cs.ru.ac.za/philip/Courses/CS3-OS/Cs3ToOS.pdf`
13. McKusick, M.K., Joy, W.N., Leffler, S.J., Fabry, R.S.: A fast file system for UNIX. ACM Trans. Comput. Syst. 2(3), 181–197 (Aug 1984)
14. Nichols, B., Buttlar, D., Farrell, J.: Pthreads programming: A POSIX standard for better multiprocessing. O'Reilly, Sebastopol, CA (1996)
15. Nieh, J., Vaill, C.: Experiences teaching operating systems using virtual platforms and Linux. In: Proc. 36th SIGCSE Tech. Symp. on Computer Science Education. pp. 520–524. SIGCSE '05 (2005)
16. Pamplona, S., Medinilla, N., Flores, P.: Exploring misconceptions of operating systems in an online course. In: Proc. 13th Koli Calling Int. Conf. on Computing Education Research. pp. 77–86. Koli Calling '13 (2013)
17. Pfaff, B., Romano, A., Back, G.: The Pintos instructional operating system kernel. In: Proc. 40th ACM Tech. Symp. on Computer Science Education. pp. 453–457. SIGCSE '09 (2009)
18. Piaget, J.: The Construction of Reality in the Child. Routledge, Milton Park (1954)
19. Pietrek, M.: Inside the Windows scheduler. Dr. Dobb's J. 17(8), 64–71 (August 1992), `http://dl.acm.org/citation.cfm?id=134643.134652`
20. Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B.: Design and implementation of the Sun network filesystem. In: Proc. Summer USENIX Conf. pp. 119–130 (1985)
21. Schmidt, A., Polze, A., Probert, D.: Teaching operating systems: Windows kernel projects. In: Proceedings of the 41st ACM Technical Symposium on Computer Science Education. pp. 490–494. SIGCSE '10 (2010)
22. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts. Wiley, Harlow, Essex, 9th edn. (2012)
23. Tanenbaum, A.: Modern Operating Systems. Pearson, Harlow, Essex, 4th edn. (2014)
24. Uhlig, R.A., Mudge, T.N.: Trace-driven memory simulation: A survey. ACM Comput. Surv. 29(2), 128–170 (Jun 1997)