Teaching Operating Systems: Just Enough Abstraction

Conference Paper · July 2016

DOI: 10.1007/978-3-319-47680-3_10

1 author:

Philip Machanick
Rhodes University
129 PUBLICATIONS **2,342** CITATIONS

SEE PROFILE

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, but without requiring the time and grasp of detail needed to understand a full operating system implementation. My finding with designing a course to meet those goals is that various strategies apply to different concepts: faking the detail in some cases, and using techniques from computer architecture simulation in others. The overall result is a course where students have a better sense of how things work than a conceptual approach without as much time required as a full implementation-based course. Working out what "just enough abstraction" is remains a work in progress.

1 Introduction

Operating systems (OS) courses divide roughly into those that provide a general survey of OS features and variations – a conceptual approach – and those that dive into detail. The latter category includes courses that are based around 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 [31]. Despite some using creative approaches like running the OS in a virtualizer [19], working with a real full-scale kernel like Linux is challenging and cannot be done in a relatively short course.

I exclude from the discussion courses about system administration, user-level training and so on – my focus is on a traditional computer science view of an OS course – learning fundamentals such as scheduling, process and thread management, memory management, inter-process communication, input-output, microkernels *vs.* monolithic kernels and device management.

At Rhodes, most of our undergraduate courses divide into short modules that run 4–6 weeks; the OS course in third year runs over 4 weeks, with 5 lectures per week and one practical (of 3 hours) per week. Allowing for timetabled contact time of 8 hours, that leaves 12 hours a week at most for students to work on their own, assuming a standard workload of two third year courses and a 40-hour week. With that amount of time available a course that goes into the detail of a full-scale or even cut-back OS such as MINIX would not cover much ground. At the

Preprint: final shortened version to appear in Proc. SACLA 2016, CCIS 642, Springer-Verlag, 2016

same time, a conceptual course has limited value. Working with implementations and solving problems provides deeper learning than any survey can. And in any case in today's world of very high-level managed-memory languages, a keener appreciation of what is underneath all those abstractions requires drilling down to the hardware layer – at least to some extent.

Since there is an increasing trend for programming courses to be taught in higher-level languages that manage memory and abstract away all the details of the machine[1], one of my goals in the course is to expose students to the machine layer and notions like machine addresses, which they only see in a small part of their curriculum.

Finally, learning really requires some exposure to how professionals in the area world work [14] – 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:

- *faking part of the system* – for example, to illustrate how a file system could work using an internal organization like a file allocation table (FAT) [7] or the venerable inode (index node) pointer structure [16], it is sufficient to implement RAM-based structures illustrating how the disk-based pointers would be organized
- *architecture simulation techniques* – trace-driven simulation [32] used to be a popular technique in computer architecture simulation before full-system simulations became viable; this approach allows a small part of a system to be simulated provided a trace of memory accesses is available to drive the simulation

The course has been running in this form since 2015, so there is only one complete run of the course on which to report; I rely on my own experience of teaching in multiple institutions (four universities in South Africa and one in Australia) to evaluate, rather than quantitative evaluation, since there is inadequate data for a quantitative study.

In the remainder of this paper, I give some background to educational approaches that apply to this course and other approaches to teaching operating systems. Based on that background, I explain the design of my course. I then provide more detail, share experience from running the course this way and wrap up with conclusions.

2 Background

Earlier theories of learning were strongly focused on the cognitive. The constructivist model, for example, inspired by the work of Piaget [24], was based on

[1] The main programming courses at Rhodes use C# and there is also a substantial course in functional programming.

different levels of sophistication of building mental models [3]. Somewhat similarly to Bloom's Taxonomy, which ranks different levels of problem solving in terms of sophistication [13].

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 [10].

The social construction model goes a step further and divorces learning from cognitive models, focusing instead on how knowledge is created by social interaction [14,5].

Whether we continue to accept a cognitive aspect to learning or change our focus to a purely social model (backed by the big guns of phenomenology [4]), there is a clear consensus that learning requires *doing* – which strongly argues against a pure survey approach to OS teaching.

Understanding an OS requires overcoming a number of misconceptions [21]; it is hard to see how such misconceptions can be overcome without a strongly practical component to an OS course.

So what is the experience of doing-based OS courses?

In the early 2000s, *instructional operating systems* such as Nachos, Topsy and Yalnix that were designed to abstract key concepts to simplify teaching operating systems had a following [2]; Pintos is a more recent design [23]. None of these however have the critical mass required to develop a wide range of course materials. Even such cut-down systems require significant overheads to learn how to work with them. In discussions with academics at University of New South Wales who work on the L4 family of microkernels [6], it became clear that even a small kernel like that cannot be learnt without a very intensive course.

There are courses that teach kernel concepts using the Linux kernel [19,11,8] and even some who teach Windows internals [28].

All of these whole-system approaches, whether using cut-down teaching systems or a commercial-scale OS, require significant time to learn the basics before getting into detail.

3 Course Design

For the Rhodes Computer Science 3 OS course, the approach I take is to cover the major concepts in lectures, then drill down to implementation in practicals. I provide detailed notes [15] and my approach is to 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 traditional OS course outline, except I cover parallel programming rather than an abstract view of processes and threads:

- *The Kernel*
  - system calls and interprocess communication (IPC)
  - what goes in kernel *vs.* user space – including the debate about 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
  - history and rationale for memory management
  - key concepts of virtual memory
  - more advanced concepts
  - examples including variations in page tables on real machines and basics of 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 operating system texts that do just that [30,29]. The challenge is how to approach these topics in more depth without a full implementation of an operating system – or more specifically, in a relatively short time.

The approach I take is to implement small fragments of an operating system that can be designed, implemented and tested independently so that a whole operating system 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 drive 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 operating system. 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 – see Fig. 1 – or FAT file system.

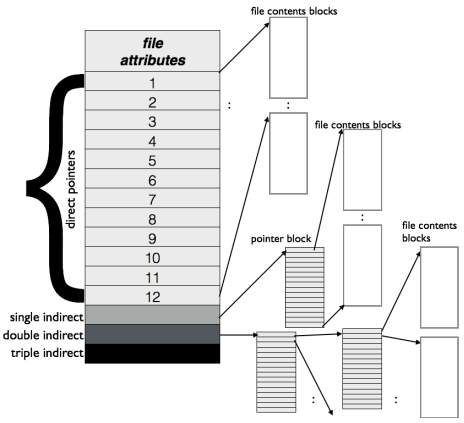

Fig. 1. Conceptual index node (inode). The top-level block contains file attributes, 12 direct pointers, an direct pointer, a double-indirect pointer and a triple-indirect pointer. Each pointer block is the size of a disk block and contains as many pointers as fit into that size.

There is much more to a file system than this, but, in keeping with the need to fit key concepts into a short time, I judge these concepts to be sufficient to give a flavour of ow file systems work. I leave out a lot of detail, such as ways of structuring directories, models of permissions and ways of implementing hard links (or similar concepts like soft links and aliases).

3.2 Trace-Driven Simulation

To implement trace-driven simulation, I generate traces using the Pin tool [12]. Pin is relatively easy to customize and I use it 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 markers where an interrupt could have occurred (artificially generated, not based on real interrupts) and each indicates the latency of handling the interrupt.

Using these traces, I can implement relatively simple simulations of schedulers and aspects of the virtual memory system. In these, I aim to illustrates principles with sufficient detail that students who understand clearly what they are doing could move to implementation of a real system without learning a lot more – other than detail.

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 this part of the course. Areas covered include Pthreads

```
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. Highly simplified inode structure. I do not include many details found in real inodes, such as timestamps and link count. The first byte must be nonzero for a valid entry; this way the VFS can detect if an inode entry is valid without knowing its internal structure.

[18], 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 virtual memory. 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 system (VFS) was originally designed to hide implementation details such as whether the file system is local or remote [27]; in my approximation to this, the top-level file system implements operations to fetch and replace blocks on a device simulated in RAM; the actual file system relies on these operations without needing to know where the actual blocks are stored, capturing the essence of the idea of a virtual file system without the complexity. This allows me to set a question in which a detail of a highly simplified inode-style file system has to be implemented.

Figure 2 illustrates my minimalist inode structure. It contains a pointer to a data structure defining the virtual file system in which it is contained. Otherwise, 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 – but does not know how big they are, and is initialized by the actual file system with these attributes.

In my notes I describe how to navigate to a particular block in an inode system as follows (P_b is the number of pointers per block):

> *To find a block number given an offset within the file is easy: we just do an integer divide. So the hard part is, given a block number find the actual block by navigating through the* `inode` *structure. Let us focus on that.*
>
> *What we are doing is translating the relative block number within a file (how far into the file we are, counting in units of blocks), b_R into an absolute (device-level, D) block number b_D, counted over all blocks on the device.*
>
> *The basic idea is that you start with the relative block number within a file, and as you eliminate part of the inode structure as being too small to contain that block number, you reduce relative the block number by the number of blocks in the earlier part of the inode structure. Here are the steps in outline, for translating relative block number b_R to device block number b_D (assuming b_R is a valid block number, i.e., not off the end of the file). At each step, if we are able to retrieve b_D, we stop. Whenever we need a pointer block, we need to retrieve it (it may be on disk or in memory, but we do not deal with that detail here). The steps in outline – stopping as soon as you find b_D – are:*
>
> 1. *$b_R < 12$? If so, is use b_R as an index into the direct blocks and retrieve b_D*
> 2. *replace b_R by $b_R - 12$. Is the new $b_R < P_b$? If so, use b_R as an index into the indirect pointer block to retrieve b_D.*
> 3. *replace b_R by $b_R - P_b$. Is the new $b_R < P_b^2$? If so, find out which second-level block b_R as in and look up b_D in that indirect pointer block.*
> 4. *replace b_R by $b_R - P_b^2$. Is the new $b_R < P_b^3$? If so, find out which second-level block b_R as in, then which third-level block b_R is in and look up b_D in that indirect pointer block.*

Understanding a description like that in the abstract is not easy for most – writing code to implement it is certainly useful for most computer science students as a way of making a theoretical formulation real.

The actual file system asks the VFS for blocks, either offset from the start of the VFS for overhead blocks, or from the start of the file region of the VFS for file content.

The VFS defines a number of other details that would be difficult to implement without knowing how the device is implemented, including a free space data structure, implemented as a bitmap.

Figure 3 illustrates the data structure for the simplified virtual file system. To keep things simple, I allocate fixed regions for directories (one per file: I do not implement a directory hierarchy). This makes it relatively easy to do operations

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

like creating a file or searching for one given its name; I avoid complications like a directory structure that would scale up to a large number of files.

These simplifications make it possible to set a question like implementing the operations on an inode-based system to create, remove or extend a file – or to do the same with a FAT system. The bitmap representing free or allocated blocks also provides an opportunity for exercises involving bitwise operations. The conceptual challenges students must deal with include understanding that file system pointers are not the same thing as memory pointers (they refer to blocks on disk, not bytes in main memory) and that the data structures used to represent files can be complex to navigate.

4.2 Trace-Driven Simulation: scheduling and VM

Pin allows me to instrument code with relatively low effort; I produce trace files that mark memory addresses as one of read ("`R`"), write ("`W`") or instruction fetch ("`I`"). I also add in fake interrupts at regular intervals, each with a fixed latency (marked as "`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
I 0xb78882a6
W 0xbfd912b8
I 0xb78882a8
R 0xbfd91564
```

In this example, there is an interrupt with latency `0x3E8` = 1000_{10}. Latency is in clock ticks, and each instruction fetch is assumed to add 1 clock tick. If I am not

simulating memory hierarchy, reads and writes are fully pipelined and therefore do not advance the clock.

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 there is only one kind of interrupt and all interrupts can only be processed once a waiting process reaches the head of the wait queue.

To simulate scheduling, it is only necessary to read in the trace file processing instruction fetches and interrupts; reads and write accesses 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.

With these simplifications, it is possible to compare variations like round-robin scheduling, multilevel feedback queues (as in Windows [25] and some versions of Linux [1]) or even the completely different approach of the Linux Completely Fair Scheduler that uses a red-black tree to find the next task to schedule [20].

Naturally these simplifications do not expose students to the true complexity of writing a real scheduler, but in the spirit of "just enough abstraction" it is possible to see what the main issues are and gain practice at coding at this level.

Here is an example of a problem: I handed out code that implements a simple round-robin scheduler and asked the class to add another level of priority for processes marked as "interactive". The interactive process queue is always scheduled ahead of regular processes, so processes in the regular queue only run if there are either no interactive processes or all the interactive processes are waiting on interrupts. In a workload, for this example, if the trace file name is immediately followed by " `*I`", that indicates the trace represents an interactive process, as in this example:

```
data/test.trace*I
data/test3.trace
```

Solving this problem requires simple C coding, and clearly illustrates the concept of wait time – time spent in the ready queue – because in a workload with a single interactive process, that interactive process will have zero wait time.

Virtual Memory Virtual memory is even harder to code at the true hardware level than scheduling, since a VM implementation has to match hardware functionality closely. Nonetheless a trace-driven simulation allows a range of implementation details to be explored. Some examples include alternative page table structures (single-level *vs.* multilevel, or an inverted page table) 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.

Asking the class, given an implementation of a single-level page table, to implement a two-level page table provides a reasonably challenging programming example with design of a new data structure, dynamic allocation and use of pointers and evaluation of the difference of memory usage versus the single-level page table. Other exercises such as implementing and evaluating the effect of a TLB provide similar levels of difficulty and insights into how a real system works.

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 using the one type of parallelism, recode using the other
- *shared memory* vs. *memory maps* – again, recoding in the other type of example illustrates how they differ
- *synchronization* – focus on a subset of types of synchronization (barriers, mutexes, etc.) and build examples based on that primitive.
- *IPC primitives* – coding using pipes adds another dimension

The practical difficulty in a short course is not finding examples, but limiting the range of topics covered to suit the time. The trade off I make is to use a subset of these examples in practical questions each time the course is run, while doing more examples in lectures.

5 Experience

It is difficult to evaluate the effect of changing from a conceptual to an implementation approach when my experience of courses of this type spans multiple institutions. I used the conceptual approach when at University of Queensland, and the student mix is very different there as compared with Rhodes. I can therefore at best offer observations based on my personal 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 of the other areas like scheduling are easier to learn conceptually. However, even so, I find that having students code an example enhances understanding. Conceptual text books present scheduling in theoretical way that has very little to do with real OS design [29]; a case study of how the Linux scheduler has evolved 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.

The fact that students battle with low-level coding concepts like pointers is not a reason to avoid these concepts: they have to learn them somewhere, and an OS course is a logical place to introduce them since the OS is the interface between hardware and software (even if some argue for implementing the OS in a higher-level language [26,17], at some point you have to map hardware concepts to code – often still in C or C++ [9,22]). An OS course also provides a useful way to show that pointers can be different at 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 style of approach – 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. My class notes [15] are in the form of a Creative Commons-licensed mini text book and I welcome additional contributions to make these notes more useful.

As a course designer, the just enough abstraction approach provides a useful alternative to either of the all or nothing extremes. Teaching from a real operating system is difficult and requires a lot of overhead before coming to grips with key concepts; a pure survey approach does not lead to deep understanding. The challenge is to work out how much abstraction is just the right amount – and that remains a work in progress.

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), http://doi.acm.org/10.1145/273133.274308
4. Berger, P.L., Luckmann, T.: The Social Construction of Reality: A Treatise in the Sociology of Knowledge. Penguin, London (1966)
5. 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)
6. Borchert, C., Spinczyk, O.: Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-program analysis. In: Proc. 8th Workshop on Programming Languages and Operating Systems. pp. 1–7. ACM (2015)
7. 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), http://doi.acm.org/10.1145/225535.225536
8. 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), http://doi.acm.org/10.1145/2538862.2538894
9. Hunt, G.C., Larus, J.R.: Singularity: Rethinking the software stack. SIGOPS Oper. Syst. Rev. 41(2), 37–49 (Apr 2007), http://doi.acm.org/10.1145/1243418.1243424
10. Kim, B.: Social constructivism. Emerging perspectives on learning, teaching, and technology 1(1), 16 (2001)
11. 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), http://doi.acm.org/10.1145/1953163.1953250
12. 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), http://doi.acm.org/10.1145/1065010.1065034
13. Machanick, P.: Experience of applying Bloom's Taxonomy in three courses. In: Proc. Southern African Computer Lecturers' Association Conf. pp. 135–144 (2000)
14. Machanick, P.: A social construction approach to computer science education. Computer Science Education 17(1), 1–20 (2007)
15. 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
16. 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), http://doi.acm.org/10.1145/989.990
17. Mitchell, J.G.: JavaOS: Back to the future. In: Proc. 2nd USENIX Symp. on Operating Systems Design and Implementation. pp. 1–. OSDI '96 (1996), http://doi.acm.org/10.1145/238721.238731
18. Nichols, B., Buttlar, D., Farrell, J.: Pthreads programming: A POSIX standard for better multiprocessing. O'Reilly, Sebastopol, CA (1996)
19. 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), http://doi.acm.org/10.1145/1047344.1047508

20. Pabla, C.S.: Completely fair scheduler. Linux J. 2009(184), 4 (2009), http://www.linuxjournal.com/article/10267
21. 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), http://doi.acm.org/10.1145/2526968.2526977
22. Peter, S., Schüpbach, A., Menzi, D., Roscoe, T.: Early experience with the Barrelfish OS and the single-chip cloud computer. In: Proc. 3rd Many-core Applications Research Community (MARC) Symp. pp. 35–39 (2011), http://www.barrelfish.org/barrelfish_marc11.pdf
23. 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), http://doi.acm.org/10.1145/1508865.1509023
24. Piaget, J.: The construction of reality in the child, vol. 82. Routledge, Milton Park (1954)
25. 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
26. Prangsma, E.: Why Java is practical for modern operating systems. In: Libre Software Meeting (2005), http://www.slideshare.net/monchix/jnode-presentation
27. 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)
28. 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), http://doi.acm.org/10.1145/1734263.1734429
29. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts. Wiley, Harlow, Essex, 9th edn. (2012)
30. Tanenbaum, A.: Modern operating systems. Pearson, Harlow, Essex, 4th edn. (2014)
31. Tanenbaum, A.S.: Lessons learned from 30 years of MINIX. Commun. ACM 59(3), 70–78 (Feb 2016), http://doi.acm.org/10.1145/2795228
32. Uhlig, R.A., Mudge, T.N.: Trace-driven memory simulation: A survey. ACM Comput. Surv. 29(2), 128–170 (Jun 1997), http://doi.acm.org/10.1145/254180.254184