Operating Systems and Architecture (CO527) Spring 2016

Introduction to Operating Systems

Fred Barnes and Radu Grigore

Why Study Operating Systems?

The vast majority of programmers do *not* work on operating systems. Given this, is it a good idea to invest time in learning about operating systems? Yes. Let's see why.

There is so much software around that no human can comprehend more than a tiny fraction of it. How come we can keep writing software without getting hopelessly confused by its complexity? How come the software written nowadays does so much more than the software written 50 years ago? There is an unlikely person who'd know the answer to that last question: Isaac Newton. He knew he saw far because he stood on the shoulders of giants. He may not know what 'software' is. But, he'd probably guess that today's software stands on the shoulders of yesterday's software, and that's how it does more.

In fact, the whole edifice built by programmers looks like a skyscraper, reaching up at dizzying heights because it is carefully built layer upon layer. Layer upon layer of abstraction. This idea, of having *layers of abstraction*, is one of the most important ideas in computer science. In a sense, the whole course is built to teach the concept of abstraction by example. Of course, one may wonder why would operating systems be a particularly good source of examples of abstraction. Fair question. To answer it, let's get back to our skyscraper. It's silent around. But, if you look up, and if you look really carefully (perhaps with binoculars), you'll see a lot of activity at the top. That's where the builders are! But — hold on — they seem to be doing something strange: Instead of continuing up more or less as you'd expect, there are several teams building in different directions. The skyscraper has branches! Why would they do such a thing?

It's because they have strong opinions. Each team of builders thinks that the skyscraper should continue the way *they* build it. Each team is unable to convince the other teams that *theirs* is the right way to build. So, each team carries on with what they think is best. Eventually, what happens over and over in history is that one or two branches outlast the others. The others either collapse, or builders simply abandon them. The truth is that no one knows which will survive and which will collapse.

What is this analogy supposed to mean? The top floors of the skyscraper represent the new software. Most active programmers work on fairly new pieces of software. This new software is built in a variety of ways, and the truth is that nobody knows which of those ways is the 'right' one. Of course, everybody has their favourite way of writing software. And, of course, everybody thinks they're right. And, of course, some of us will be 'proved' right — it's inevitable. Alas, that doesn't necessarily mean that some of us are better at predicting which new software will pass the test of time.

But, if we look into the past, we can *see* which software passed the test of time. Operating systems are at level 1 of the skyscraper. Immense volumes of software are based on them, and they didn't crumble. Something about them must be right.

What is an Operating System?

Operating systems are programs that sit between hardware and user applications. This means that applications can afford to ask fairly high-level things, such as 'give me 1 MiB of memory where I can store data'. The operating system hears such a request and takes care of all the nitty-gritty details necessary to make it happen on some particular hardware.

Of course, the above definition is rather vague. In fact, it's vague enough that almost everybody agrees that it is a fair characterisation of operating systems. We can try to be more concrete, at the risk of alienating some friends, who'd suddenly think we have a too narrow definition. Nevertheless, let's try.

POSIX is a standard of what the operating system should offer applications. It lists C functions, command line utilities, and is in general rather comprehensive. We could define a (POSIX-compliant) operating system as being a piece of software that implements the POSIX interface. But, POSIX lists for example the malloc function, and some people would disagree that malloc is implemented in the operating system. Why? Two reasons are political, another is technical.

Let's mention the political reasons only briefly. First, Windows implements only a small subset of POSIX, and it is harsh to say that Windows is not an operating system. Second, on Linux systems, the POSIX interface is implemented by two antagonist developer camps: the kernel developers and the GNU developers. The kernel developers think the operating system is *only* the kernel; the GNU developers (e.g., Richard Stallman) disagree.

Now back to the technical reason. The processor can run in two modes, typically called kernel and user. In kernel mode, the processor does what it is told. In user mode, the processor refuses to do some things that look odd. For example, if the code tries to access some memory it should not access, then the processor interrupts the execution, and jumps to a piece of code that handles such an 'exceptional' situation. Some people, like the kernel developers, prefer to define operating systems as follows: the operating system is the code that runs in kernel mode.

In this course we will look at some guts of the kernel, but also at some guts that spill out of the kernel. In other words, we stick to the POSIX definition.

Content

The operating systems section of the course starts with kids' stuff — this lecture. But, don't worry, we get to the meat soon. We will study how the operating system acts as a layer of abstraction on top of four different types of hardware: memory, hard drive, network card, and processor. Each of these components will get two lectures. And for each of these components we'll study what interface does the hardware offer, what interface does the operating system offer, and how does the operating system manage to translate from one interface to the other. All in all, you'll see nine lectures about operating systems.

- Lecture 1: introduction
- Lectures 2 and 3: memory
  - interface offered to applications (malloc, free)
  - implementation in the operating system
    - memory allocation (free list, first fit, best fit)
    - caching (hierarchy; FIFO, LRU)
    - virtual memory (page table, thrashing, TLB)
- Lectures 4 and 5: hard drive
  - interface offered to applications
    - tree structure, permissions
    - files (open, close)
    - synchronous I/O multiplexing (select)
  - implementation in the operating system
    - disk blocks, free lists
    - i-nodes
- Lectures 6 and 7: network card
  - interface offered to applications (socket, connect)
  - implementation in the operating system
    - basic protocols (IP, TCP, UDP)

        * traffic shaping and queuing (SFQ)
        * congestion (CUBIC)

- Lectures 8 and 9: processor
  - interface offered to applications
    * processes (fork, wait, pipe)
    * threads (pthread_create, pthread_join, pthread_rwlock)
    * scheduler tuning (setpriority, sched_setaffinity)
  - implementation in operating system
    * basic schedulers (FIFO, round robin, priority based)

Organisation

- Lecture notes are on Moodle 24 hours before the lecture, or earlier. Slides and code may appear later. For example, slides may appear just before the lecture, and code may appear just after the lecture (because it will be written *during* the lecture). **Warning**: For this half of the course, you cannot read only the slides, you *must* read the lecture notes as well.
- There will be exercises in each set of lecture notes. We will do some of them together in lectures. Try to do the others on your own.
- We will have regular quizzes in lectures. These are not for assessment — they are for the lecturer to figure out whether some concepts need to be explained further.
- In lectures, you will see C code. The main reason is that POSIX is defined partly by C function signatures. However, (1) if you are comfortable with an imperative language like Java you should be fine; and (2) the exam does not involve C code at all.
- Sections marked* with an asterisk are optional.
- Important exercises are marked with ►.
- If you find mistakes in the lecture notes, including typos, send an email to rg399@kent.ac.uk. If you have suggestions for improvement, send an email as well.

Exercises

1. ► Read 'Why Take an Operating Systems Course?' by J Regehr.
2. Google 'why study operating systems?'
3. Find a good reference for POSIX, on the Internet.
4. Download the source code of the Linux kernel.
5. Write a 'hello world' program in C.
6. What are operating systems?
7. What is POSIX?
8. ► Get familiar with the command man, including its -k option.
9. ► Read 'Understanding User and Kernel Mode'.
10. ► Study the anatomy of the Linux kernel.