Operating Systems

Professor Richard Harvey FBCS

31st May 2022

You sit at the keyboard ready to write your magnum opus. All is quiet — the peace of the blank page. Your computer waits unobtrusively, like a sort of electronic Jeeves, ready to leap into action should you command something. But underneath that passive exterior lies a maelstrom of activity - hidden from view is an intricate command and control structure - an electronic Court of Versailles scurrying around making sure the computer can do its business and that you, the user, are treated like Louis XIVth. All that hassle and bustle, the stuff that links your application to the computer hardware, that's what's called an Operating System.

If you prefer a more prosaic example then I refer you to that radical magazine, The Beano, and the strip "The Numskulls" - imaginary little characters beavering away in the human brain and running the show. The numskulls are also just like a modern operating system - each process has its own function, and they communicate keep the brain working.

Early computers had no operating systems. The user, the programmer and, often the inventor, were the same person and if they wanted the computer to do something, then they sat in front of it and fiddled around with switches — that was programming. But it soon became evident that computers could do pretty useful things and people were clamouring to use them. But they were fiendishly expensive, so sharing was necessary. Thus, the first operating systems were methods for humans to share the computer. The arrangement was known as the *open shop* model and in [1] George Ryckman recalls the operation of the one of the first IBM 701 systems - users had a fifteen-minute slot, most of this was wasted with setting up the input and output devices which led to inefficiencies. Furthermore, the IBM 701 had a mean time between failures (MTBF) of around 3.5 hours, so programming was laborious.

The SHARE operating system took the dramatic step of removing programmers from the computer room - hence a *closed-shop* system. Programs were prepared offline and assembled onto to tape in a *batch* (hence batch processing). Batch systems made better use of machine time, but they were no less irritating for programmers: one would prepare the program then wait for hours for it to be added to the tape; then the output would be returned to discover that the program had terminated due to some trivial error. One the one hand we want to use the machine efficiently and on the other we want users to have seamless access to the machine. These two requirements pull in opposite directions and the operating system has to resolve a solution. As operating systems are complicated, it's easy to get just in the weeds and wild grasses of interrupt handlers and scheduling. Fortunately, one the gurus of Operating Systems, Per Brinch Hansen has described some innovations with the delightfully sweeping phrase of "ideas of fundamental interest" which he describes in [2] as: open shop; batch processing; multi-programming; time-sharing; concurrent programming; personal computing and distributed systems. We have already covered the first two which are now somewhat arcane, but the remainder are still part of a modern operating system.

The original driver for multi-programming was the observation that peripherals were slow. So, if the program contained the command print('HELLO') then there would be seconds of delay while the print head was configured to print the letters H,E,L,L and O. Meanwhile the computer was in wait states essentially doing nothing. The solution came in three parts: design peripherals with a modicum of intelligence so it can complete its task without intervention from the main computer; allow peripherals to signal back to the computer that they need assistance (these are often called interrupts); have a scheme that allows a computer to switch tasks. This latter aspect, although originally designed to allow computers to handle slow peripherals, soon became adapted to the more general problem of running more than one task at a time.

A modern "context switch" involves several different processes - let's look a case. The computer has instructed to the hard disc to retrieve a number held at particular address. Modern hard drives are smart and they figure that although it has been asked for the contents of one address it may was well retrieve the whole cylinder and put it in fast electronic memory - this is called *caching*. Having read the whole cylinder it sends a signal to the main computer, an interrupt, to say that it has the data. Sometimes interrupts have to be handled immediately - they have a high priority others can wait. The interrupt handler decides which interrupt matters the most and tells the main computer to stop doing what it is doing which is itself quite a complex process.

Imagine you are kitting a pattern, you are following the knitting pattern, you have a record of where you have reached in the pattern (this is what we call the Program Counter (PC) in the world of computer hardware) and you have various records of how many rows of various yarns have been used (variables) and of course you have the pattern and the pieces created so far (the program and the output data). But now it is time to make supper, so you clear your table and carefully place the pattern, the knitting and records of the variables and PC on a side table. Now we start to make our Sheppard's pie. The table has the recipe, a note of where we are in the recipe (the PC), the ingredients and tools for making the pastry and bits and bobs. "Waaaah!" eek a high priority interrupt, the baby is crying. Quickly we place a small table on to of our knitting table, we place all of half-made Sheppard's pie on that table and up onto the table comes the bawling baby. We feed the baby - service the high-priority interrupt. But baby back in the crib and we take off our "stack" of side tables the pie and continue. Ring, ring goes the telephone "Where is my jumper you were making?" asks an unhappy customer, "We make a note to work on that later" this is a low priority interrupt - we will pop the stack later for that job.

We just about handle this with some additional hardware, the interrupt handler but the system clearly offers potential for switching between tasks that are not all screaming babies. Maybe we want to read a book while preparing supper and we also want to make-ready our porridge for breakfast tomorrow. Actually, humans do not do well under such situations and high-level cognitive tasks cannot easily be switched. Computers also take time to switch tasks but there is a stronger incentive to do so — if our computer is fast enough then it can switch between multiple tasks so quickly that it appears as though each task is running concurrently. When you move the mouse across the page of your computer is it obvious that the computer has stopped all its other calculations to move the mouse a tiny bit and then switches back? Usually not. And even bigger incentive is to run multiple tasks from multiple users — we now have a multi-tasking operating system. Let's have a look at the programmer's model though.

Here is a simple program

```
A = 2
input B
print B+A
```

This is a program written in a high-level language. To make it useable by a computer we convert it into low-level instructions using another program known as a compiler. The compiled output is often a bit difficult to decipher but it a low-level version of our program might look like this:

```
STA &1000 2
LDA &50, &1001
print @1000+@1001
```

The first line reads, store the value 2 at the address in memory 1000. The second line reads load in the value from found at address 50 (I'm assuming this is where the keypad inputs its numbers) and move it to address 1001. The final line, which would normally take a few hundred low-level operations, reads add the contents of address 1000 (that's the value 2) to the contents of address 1001 (that was where we stored the input). This compiler has made some choices - that the variables will be stored in a block of memory starting at address 1000 and that the keyboard input will be found at address 50. But what if we have another user using a different keyboard. What stops them reading my keyboard? Or over-writing my program? Multi-tasking operating systems bring with them major challenges of security and synchronisation. Maybe we allow users to write programs only with a special compiler that forbids users from doing dangerous things? That approach has been tried for experimental systems, but it is not flexible enough for commercial use. Instead, the usual solution is design special hardware that has some special instructions, *protected instructions,* that can only be used by a *super-user*. This is an important concept - with multiuser operating systems comes the class-system of the computer world: super-users who themselves are often classified into various classes. And hence, as with all class-systems there are in-jokes told about users by operators and about operators by users. With privileged instructions we can stop a user's program even if they do not wish it to stop. But we still have the problem of memory overlap.

Memory clashes are prevented by telling programmers that they have a clean-sheet to store variables where they wish. In reality, those addresses are passed through a secret look-up table to map the programmer's model into reality. And reality could be quite messy - our real memory might have some internal variables which users cannot see, then a block of storage from user 1 and then a block from user 2 and then another block from user 1 and then, after we have run out of memory, perhaps user 2's memory is stored on a hard disc somewhere — the operating system having gambled that user 2 doesn't seem very active and maybe we could *swap* it out of main memory to give more working space to the system and user 1 who looks like he will need it. The convention is that these blocks of memory are made up of *pages* which are conveniently sized blocks for that particular machine. Large pages mean users have large blocks of physical memory available but use up much memory so are expensive for large numbers of processes. Small pages mean inefficient swapping in and out and can lead to highly fragmented memory which in turn leads to wasteful memory address calculations. By writing pages out to disc, using tasks which hopefully run in the background without anyone noticing, we can provide programmers with an impression of almost limitless memory - virtual memory. Unfortunately swap pages in and out of physical memory is a slow process and under certain conditions the machine can be brought to halt by the act of paging memory - an undesirable condition known as *thrashing*. When physical memory was very expensive, it was commonplace to rely on virtual memory and thrashing was commonplace especially on the smaller varieties of VAX computers, which dominated academia. Hence the parody of the Lewis Carol poem was often found pinned to the wall of the machine room:

Speak roughly to your little VAX,
And boot it when it crashes;
It knows that one cannot relax
Because the paging thrashes!
Wow! Wow! Wow!

I speak severely to my VAX,
And boot it when it crashes;
In spite of all my favorite hacks
My jobs it always thrashes!
Wow! Wow! Wow!

Once these developments had been realised, it became possible to write operating systems of the form that we know them today. However, a recurrent problem in the early days of operating systems was size — all these features made the operating system a large monolithic block of code. As the machine booted up the loader, loaded the ginormous operating system and there was no real memory space left for users. It was analogous to modern university in which a preponderance of government rules means that there are three administrators for every teacher, and we spend more time administering the university than we do teaching in it. There are administrators who do nothing but administer administrators. While universities seem to find this congenial, computer scientists do not like it and hence the birth of the kernel-based operating systems. The idea is write positively the smallest program possible to administer the computer — other more sophisticated functions are written as separate programs which are called when necessary. This is efficient in terms of space, but it brings two potential problems - the first is change control and standardisation. This is a familiar problem in mechanical engineering — will a BMW 7 series gearbox from 1985 fit into my 1986 9 series? We tackle this by logging development software in and out of a code repository so we can not only track changes but roll-back mistaken changes. The second solution is borrowed from electronic engineering - standard components and interfaces.

By standard components I mean *libraries*. Even in the very earliest days of computing it was soon obvious that there were certain mundane operations that everyone needed to do - print; add two complex numbers; read something from a tape and so on. Obviously one route, which is popular with undergraduates, is to copy code from stackexchange and hope the Professor does not notice. But for a commercial developer this is a nightmare - when Bloggs who wrote the code updates it, how do you know? A further challenge is that when we, say, install a new printer, we all have to write code specific to that device. So early operating systems had hundreds of *drivers* which were specific bits of code and which often had different programmer's interfaces. The solution is to define a virtual printer with a standard interface and then to write code that allows conversion between that interface and reality. This means all programmers are now writing for the same thing - a virtual printer and their code is shorter because they are including mountains of computer code. This problem is particularly acute when you think about architectures that encourage multiple hardware manufacturers – there are hundreds of types of Android phone. In cases such as these, much effort is expended in *abstracting* the hardware. Indeed, Android has a special interface known as the Hardware Extraction Layer of HAL[1].

Because we are dealing with virtual memory in the user code, and possibly absolute addresses in the libraries, it is commonplace to compile the user program to nearly machine code (*object code*) but with some of the addresses left free then to link in the libraries and resolve addresses to form an *executable*. Libraries can be horrendously large - I remember writing a program that wrote "Hello World!" on an early windowing workstation and being astonished to discover the linker added over 100,000 lines of code to handle the windows. Hence dynamic libraries are the norm in which only relevant parts of the library are used and executed.

We have talked about how multiple users and multiple processes can be managed – the computer context switches and allows some time to each one of the important tasks. But there is much important detail in task *scheduling* and much ink has been spilled on varieties of scheduling algorithm. In the lecture I use one from Solaris which is a form of Unix developed by Sun microsystems and now maintained by Oracle. The basic ideas are as follows:

1. Each thread[2] is assigned a class and a priority.

[1] This is a little computer science joke – HAL 9000 was the name of the computer in 2001: A Space Odyssey
[2] The word *thread* has snuck into this discussion without much drama. A thread is a task that can be run as a separate process. When a programmer writes a program they may write it as a single sequential stream of instructions – a single thread. Or they may realise that there are multiple threads that could run in parallel.

2. The CPU runs a clock which slices time in discrete intervals (20ms is the default setting for Solaris)

3. Most threads are allocated as time-sharing threads in which the higher the priority, the more likely it is to be scheduled but the less time it is allocated.

4. One a thread has exceeded its allocated time; its priority is lowered thus hungry tasks are not allowed to “eat all the sandwiches”

One interesting class of processes are those that are classified as *real-time*. The phrase real-time is often misused to mean jolly fast but it has a technical meaning which is a process for which we can guarantee a response within a fixed time. An example of a real-time process is the echo-cancellation task which forms part of a modern video conferencing software – if the echo cancelling filter is not adapted quick enough then there is howl-around, and everyone has to turn off their microphones. Different computers have different processing power so you may have noticed during the pandemic that users with dated machines could cause whole meetings to be disturbed. Operating systems that specialise in Real-Time responses are often abbreviated to RTOSs and are often found in expensive systems such as stellar explorers.

A scheduler is just one of the components that can affect how a machine runs code. Another is IPC or inter-process communication. Sometimes a programmer’s code can run using very little I/O – an algorithm to compute if one number is exactly divisible by another reads in the numbers, computer, computes, computes and gives the answer “yes” or “no”. Such is a *CPU-bound* process. More likely there is much interaction with the real-world and it is *IO-bound*. It’s obvious that an IO-bound process will have to wait for the peripherals and devices to do things – the device drivers and the program have to exchange signals – wait while I read this character – tell me when you are ready to receive my next block of audio samples and so on. But CPU-bound processes communicate too – especially if they consist of multiple threads.

While inter-process communication is essential inside lurks many a danger. Let’s imagine we have two processes running in parallel – let’s consider two processes operating in parallel. Each process reads a global number, a count, and increments it by one. So, if we run process after another our count would go 0, 1, 2. But let’s say the second increment starts a bit early – maybe we have an optimising compiler which has automatically parallelised the process and it starts the first incrementor before the second is finished. Now the count goes 0, 1 but the second process started before 1 had been written out so it also reads a 0, adds 1 and now we have 1. The count reads 1 even though we have run the increment twice. This is called a *race* condition because it can lead to never ending loops. Let’s imagine we wanted to terminate when our count reached two – in the case above we would never terminate. Nasty.

Good inter-process communication can help minimise the possibilities of deadlocks and races but it also requires some skill to anticipate the way in which the operating system might choose to run your tasks[3].

So far, we have been largely describing the historical situation as it existed a few years ago. But the most recent trend is what is loosely called *cloud* computing. Cloud computing comes in a variety of flavours. At the top of the hierarchy is a form known as SaaS or Software-as-a-Service. If you use

Furthermore, a modern compiler might well decompose your single thread into multiples for speed. As we saw with context switching there may be quite a lot of overhead associated with switching between full processes but threads, because they are children of processes, can be more *lightweight* meaning that the processor has to save less stuff when it switches threads.

[3] A famous example of a block happened on the 1997 Mars Pathfinder mission – a long high-priority task was forced to wait for a low-priority task to release some resource. One of the tasks took an unexpectedly long time to complete and everything fell in heap. Fortunately, the system had been safely designed so it realised there was a problem and rebooted. Engineers were alerted to the problem by frequent reboots. The whole legend is described in [4].

the online version of Word or Google Docs then this is SaaS. Of course, it doesn't entirely run in the cloud – when you use the online version of, say Apple's Pages, then you are downloading a program written in Javascript which runs in your browser. It can access your local files but only in a strictly controlled way – the application runs in *sandbox* environment that blocks any attempt to access naughty things. More difficult to describe but more powerful is PaaS or Platform-as-a-Service. Examples of PaaS systems include Amazon's Web Services, Microsoft Azure, IBM's cloud and so on. Whole systems for doing machine learning, advanced computation and large-scale storage plus development tools are all available remotely. And beneath this level we have Infrastructure-as-a-Service – remote machines that are *hosted* by a provider and are available for your use. Commercially these are quite popular for several reasons, among which are the cost of running a secure server room and the possibility of hosting servers in locations with plentiful green electricity. These may be physical machines owned by the customer (or rented) but more likely now they are virtual machines.

Virtual machines are operating systems which run inside a sandbox which captures all the hardware calls and reroutes them to appropriate hardware. The software that runs all these virtual machines is also an operating system (sometimes called a *hypervisor*). Type 0 hypervisors often use specialist hardware but more common now are Type 1 hypervisors which are specialist operating systems (or in the cases of Red Hat Unix and MS Windows Server general purpose operating systems with some add-ons). Not only does cloud computing offer financial benefits to corporate customers, it can also offer considerable flexibility – as your commercial service scales-up one can spin-up some more virtual machines and, in the case of the UK government at least, one can procure virtual machines under a standard contract [3].

In this lecture series we have looked at inventions which are enormously important to the modern world but seem to be understood by only a few people. To most people the Operating System is the "look and feel" or the Graphical User Interface but, as we have seen in this lecture, that is just the surface. Underneath that serene exterior there is an electronic bedlam of context switching, memory shuffling, input/output and constant evaluation, and optimisation. Operating Systems have a reputation for being incredibly difficult to write so it is small wonder that there has been a considerable convergence and, ignoring embedded and industrial systems, we are now down to three options: Windows (which has been so several different incantations – the current one, Windows 11, being based on OS/2); Unix (of which there are zillions of variants – the most prominent being MacOS/iOS/watchOS which is based on Mach and Linux (which almost comes in multiple versions but is dominated by Android which dominates the mobile device market). Given that Linux is a Unix rewrite it looks very much as though one OS has dominated. Unix was conceived by two chaps working at AT&T Bell labs – the same labs that developed Information Theory, the transistor and the telephone. Quite a record!

References & Further Reading

1. G. F. Ryckman, "The IBM 701 computer at the General Motors Research Laboratories," *IEEE Annals of the History of Computing,* vol. 5, no. 2, pp. 210--222, 1983.
2. P. Brinch Hansen, Classic Operating Systems: from batch processing to distributed systems, new York: Spinger, 2001.
3. A. Mukhametzhanova, R. Harvey and D. Smith, "Ahead in the G-clouds: policies, deployment and issues," *Electronic Government and the Information Systems Perspectives EGOVIS 2014,* 2014.
4. M. Jones, "What really happened on Mars Rover?," 07 December 1997. [Online]. Available: http://eos.cs.ovgu.de/eos_old/lehre/WS0708/vl_pkes/folien/risk.pathfinder.pdf. [Accessed 24 May 2022].