EFFICIENCY : GETTING THE MOST OUT OF THE MACHINE.

The goal of increasing the amount of work which could be done by the expensive hardware was the original stimulus for the development of operating systems. In this chapter, we emphasise the stepwise nature of this development, each step marked by the identification of some centre of inefficiency and its eradication. This evolutionary process characterised the first half of the development of operating systems. Of course, this is an idealised account; things are rarely as neatly organised as our description, but the overall picture is sound. We enlarge on this brief summary throughout the next few chapters by describing the technical developments in more detail.

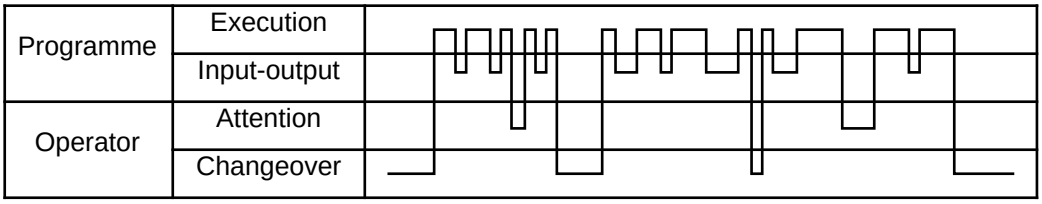
HOW IT ALL STARTED.

Once upon a time, there were only computers. By modern standards they were computationally tiny, though physically huge. If you wanted to use one, you had to know all about it, and you had to talk to it essentially in binary (though sometimes you could use equivalent, but totally incomprehensible, character strings). It was up to you to load your programme, and run it, and to fish about in its innards to conduct a post mortem if need be.

THE NEED FOR GREATER EFFICIENCY.

That was good fun, and all right so long as no one was trying to make money. But once computers started being used commercially, people soon noticed that the programmers spent a great deal of time fiddling about with the machine with no very obvious result. That was the beginning of an evolutionary process which, though different in detail, is still continuing.

The aim of the evolution has always been to bring computer services to people more effectively – generally, by devising some way to make the system work more efficiently. To find the inefficient steps, we should study the overall process. When we do so, it turns out that we can represent it, as it was around 1950 or so, something like this :



Here, the line represents the progress of work through the system, showing how the current activity switches between the programme performing internal computing and input or output, and the operator giving attention to a running job (maybe changing the printer paper or mounting a magnetic tape, for example) or changing between successive jobs.

The diagram is obviously purely schematic, so don't look for precise measurements. And don't believe the time scale at all; if we presented that precisely it would have been impossible to draw a comprehensible diagram. Very approximately, you can suppose that unit distance along the time axis represents microseconds, milliseconds, seconds, and minutes reading down the diagram from the top line – so, for example, one of the periods marked at the changeover level might easily be sufficient to complete the execution of several small programmes.

How does this help us to find the inefficiency ? In one sense, the diagram shows no inefficiency at all; at every instant, something is happening to push the computing work through the system, so from the point of view of an individual job (assuming that jobs have points of view) no improvement is possible. If we want more efiiciency, then, we can't get it by shortening the lines in the diagram; our only recourse is to the possibility of *overlapping* lines in the diagram by trying to do two (or more) things at once.

We have discovered what is perhaps the most powerful tool for increasing the efficiency of operating systems. It works at all levels, from steps in the machine

instruction cycle up to the scheduling of complete large jobs, and many of the advances of the first decades of operating systems depended on exploiting overlapping of one sort or another. It works because in many cases the agents performing the tasks – the computing, the input and out, and the administrative jobs – are separate, or can be separated, and can therefore be run on different tasks at the same time.

COMPUTER OPERATORS.

As is clear from the IBM1620 instructions, a lot of the work was fairly non-technical – loading cards and tapes, looking after printers, and so on. It became clear that if someone skilled in these tasks were available to do them, they could be done a lot more quickly. So *computer operators* were invented. They made a very considerable difference – particularly if the programmers were kept well out of the way. The next step was to try to make the operators' job more efficient.

BATCHING.

The slowest process in the diagram is the changeover. Is there any way to reduce the time spent on loading new programmes and clearing up at the end of a programme's run ? We can't change the set of tasks which have to be done to complete the work which comes in, but we might be able to change the order. Suppose that several tasks require the same programme – a sort utility, or a compiler, or an accounting programme, for example. The tasks could be distinct computations from different people's jobs, or separate transactions which pertain to the same job but arrive over a period of time. If the operator is clever enough, it's possible to collect these together and run them through the required programme as a single *batch* without reloading the programme. It needs very good organisation, because the parts for the different jobs have to be sorted back into their own jobs correctly, but it works.

Even so, it's an expedient rather than a solution. The organisation we mentioned is tricky, but essentially routine and repetitive, as is much of the other work done by the operator. Computers are designed to do routine repetitive jobs – so why can't the computer do these operator's jobs for itself ?

A SIMPLE MONITOR SYSTEM –

It can; it did – but at a little cost. First, there had to be programmes – the *monitor system* – to do the routine jobs, and they couldn't just be kept in very expensive primary memory all the time, so you had to have a disc drive. (There were magnetic tape versions, but they weren't too good.) Second, a little bit of the monitor system had to be kept in primary memory all the time to deal with things that happened in the programme; we'll say more about this *resident monitor* later in *A MONITOR SYSTEM*. That meant you lost a portion of your precious memory. Third, instead of following instructions like those for the IBM1620, the programmers had to put instructions to say what had to be done into their (usually) card decks – that's the first sign of a *job control language.* And, fourth, the programmers could no longer expect to take over the computer (except perhaps at 3 a.m.) to sort out their problems.

– AND ITS DEFICIENCIES.

If you were a programmer, the reduced access to the computer was a deficiency. (If you were an accountant, you disagreed.) Everything now had to pass through the operator, whose job was to keep the flow of work running smoothly – which it did, by and large, so long as all the programmes worked. There was trouble when programmes didn't work, because there wasn't enough room for a resident monitor system big enough to do anything but straightforward routine tasks. It was still up to the operator to sort out such problems.

And there was still a lot of spare time. When a programme was waiting for the operator to load a tape, and often while the processor was occupied with input and output, no useful computing was happening. Particularly in commercial establishments, where big files on tape were common and most of the work was largely input and output with a little bit of computing, that was seen to be wasteful.

A MORE ELABORATE SYSTEM.

There are two possible ways to go : either you can try to get rid of the spare time, or you can try to use it. A common way of cutting down the expensive spare time was to move it to another, cheaper, computer. By restricting your main computer to input and output on some fast device (a tape drive) only, you could reduce the waiting time to a minimum; then you could write the input cards onto the tape and copy the output from tape to printer using comparatively cheap computers. This is called *off-lining*, and it works. Indeed, the idea is still around, but now usually appears in the form of special processors, often called channels or device controllers, to drive peripheral devices.

The only way to make use of the spare time is to find the processor something else to do. If the running programme has to wait, then the only possibility is to run another programme, so *multiprogramming* was born. This required ways of managing several programmes in a computer at the same time, and of switching the processor between them when it would otherwise be idle. This is getting too complicated, and a good deal too quick, for an operator to handle, so the computer has to do it. Suddenly the monitor system stops being just a handy piece of software which helps when one programme comes to an end and the next one starts; it becomes a much more sophisticated thing, which manages the computer's resources, and is active all the time. It is growing up into an *operating system*.

OPERATING SYSTEMS.

The development of operating systems was greatly helped by the rapidly decreasing cost of hardware. The early operating systems (say, 1960 to 1970) grew up while hardware was still expensive enough to make it worth using efficiently, but not so expensive as to make it impracticable to reserve a certain amount for use by the operating system.

The first systems were designed for *"closed-shop"* operations, with all details of work to be done first encoded (usually) on cards, then run without any direct contact with their owners, after which the output produced would be printed and returned. Such systems became known as *batch* systems. They have little enough connection with the idea of batching we mentioned earlier; perhaps the name stuck because you had to collect all the instructions for your job into a single deck of cards, rather than running them individually. Whatever the reason, the new meaning of batching supplanted the old, so that batch processing is now usually understood to mean "without interaction with the person using the programme", as contrasted with "interactive".

The interactive systems followed; but this is about where the impetus of hardware efficiency began to run out of steam. Again because of the decrease in hardware costs, it became clear that they no longer dominated the overall cost of computing; comparatively speaking, people were getting much more expensive, so the aim switched to using more hardware if necessary to get the most out of the people. We'll say more about this in a later chapter.

AN OVERALL VIEW.

Here's a table which summarises the effects of the steps we've mentioned, and a few more which will turn up later. The main object of the exercise when these changes were being developed was to reduce the time spent on the three slower activities we showed in the diagram above so that the system would spend a greater proportion of its time processing. The table lists the various changes and any additional resources needed to implement the changes, then marks the locus of the increase in efficiency. The final column is an advertising ploy to give you the impression that it all works.

Change		OPERATORS		PROGRAMME	
	Needs	Changeover	Attention	Input-Output	Processing
Operators	Running instructions	●			▪
Batching	Clever operators	●			▬
Off-lining	Other computer			●	▬
Interrupts	Special hardware			●	▬
Disc	Disc		●	●	▬
SPOOL	Disc			●	▬
Monitor	Language, conventions	●			▬
Multiprogramming	Operating system			●	▬

And, indeed, it does work – but (as in real evolution) the development was by no means as orderly as the diagram suggests. There was much overlap between the different developments, and different manufacturers followed different paths in parallel. The cumulative effect, though, is right, and in hindsight this is a good way to rationalise the evolutionary process.

COMPARE :

Lane and Mooney[INT3] : Chapter 2; Silberschatz and Galvin[INT4] : Chapter 1.

QUESTIONS.

Why isn't the old sort of batching much use any more ? Consider the times involved in different steps of the operation.

Why did people use off-lining ?

Why does off-lining work ? Try working through this example.

Illustrate what is meant by "off-lining" by describing the sequence of operations needed to process a data file F, presented on punched cards, through a programme P run on a large fast computer to produce a result file R, printed on paper. Call the large fast expensive computer LFE and the small slow cheap computer SSC. Also describe briefly how the job would be handled by a spooling system.

Write down expressions for the cost of each of the steps in the operations described, using these symbols for the quantities involved :

Size of F :	I kB
Size of R :	O kB
Speed of card reader :	c kB s^{-1}
Speed of line printer :	p kB s^{-1}
Speed of tape drive :	t kB s^{-1}
Processing speed of LFE :	L kB s^{-1}
Running cost of SSC :	S \$ s^{-1}
Running cost of LFE :	R \$ s^{-1}

Assume that :

- The "kB" used to calculate the LFE processing time is the sum of the input and output file sizes (I + O).
- No time other than input and output time is spent on SSC.
- The programme in LFE reads its data directly from the input device and writes its results directly to the output device. (Yes, it's unrealistic, but I don't suppose you want additional complications from disc speeds on LFE.)
- There is no overlapping of input, output, and processing.
- The speed of the devices is independent of the computer to which they are attached.

If the device speeds are given by c = p = 0.5 and t = 50, find the condition for off-lining to be cheaper than spooling.
