ONWARDS AND UPWARDS – OPERATING SYSTEMS

WHAT HAPPENED TO MONITOR SYSTEMS.

How did the monitor systems change as they grew up into operating systems ? Some of the changes are listed here, under the headings used in the chapter *WHITHER MONITOR SYSTEMS ?*. Notice, though, that many items could be listed under more than one of the headings – for example, the growth of administrative and accounting functions required not only more space to hold the requisite data but also good security to keep them safe. This in itself points to an important development : the many separate functions of the monitor system are becoming more integrated into a single large entity.

MORE SPACE.

Bigger primary memories make it possible (obviously) to run bigger programmes, and (less obviously) to fit several programmes at once into the computer. Bigger secondary memories, particularly discs, provide space for the data files required for administrative purposes; it is also possible for people to keep their own data in disc files. We therefore find that *user information* systems (administrative systems, including programmes to handle functions such as accounting and security, and files holding personal information on all the people entitled to use the computer system), file directories, tape directories become prominent; administrative software to record and regulate people's access to the system becomes necessary.

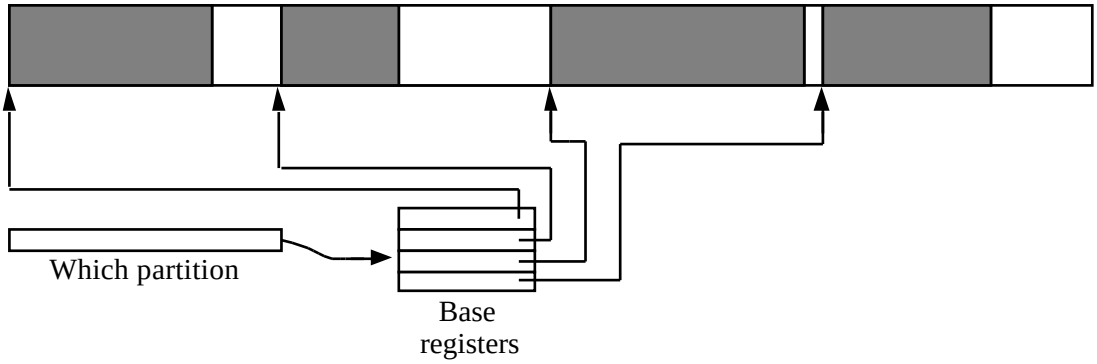
SECURITY.

A security system is now essential : people must be able to share the system resources without inadvertently sharing each other's memory space and files. We need protection for our property – disc files, tapes – against other people; the system must be protected from everybody. But protection isn't a binary attribute, which would be comparatively easy to implement. Instead, there are several different sorts of protection, all of which are needed : for example, people must be able to execute a programme, but not copy it; to read a file, but not change it.

MEMORY MANAGEMENT.

Each programme is allotted its own area of memory, out of which it may not stray. The operating system might have its own working memory, distinct from any of the ordinary programmes' areas and subject to the same protection mechanisms; in addition, some parts of the operating system must be able to manipulate any part of the real machine memory.

It is impracticable to impose such constraints on the system without hardware assistance. (It's possible in software, but the overheads are enormous – though it would be quite feasible to implement such a system on modern hardware, and still end up with a running speed much faster than that of the raw hardware of the middle 1960s when these methods were being developed ! If that were not so, Java wouldn't be very usable.) We can begin by implementing a hardware version of partitioning (*MEMORY TRICKS*). The hardware must include means of remembering the origins of the partitions, and their ends (or, equivalently, their lengths), and also means of checking that all addresses generated by the programme were in the permitted range. In effect, and possibly in fact, the processor contains a set of *base registers*, each of which holds the address of a partition :



There might also be *limit registers*, holding the last addresses in the partitions, or *extent registers*, holding the sizes of the partitions, though in a system with a constant partition size that can be built in. Given this machinery, a next step is obvious : we can make a system in which programmes can be run in any partition by compiling the programmes so that they always pretend that they begin at address 0, and building the hardware so that it adds the current base address to every address produced.

Does this help the big programmes, which couldn't fit into the partitions ? Yes – particularly if we move on to the next step, which is to make the various registers writable, so they can be fitted to the current demand for programme space. This change is not quite as beneficial as it sounds, though, because if you can put your programmes anywhere you run the risk of having a biggish one stuck in the middle of memory and leaving no room for any others. Still, the flexibility is good.

STANDARDS.

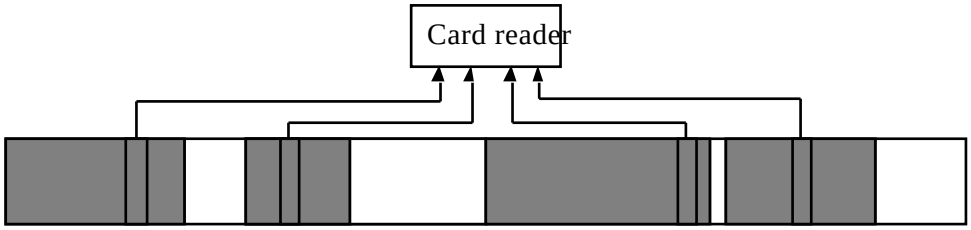
The importance of standards was not appreciated until much later. Perhaps this was a consequence of the continuing emphasis on making the machine more efficient. The unfortunate result was the growth of many different, and quite incompatible, pieces of software. Each package would be equipped with its own file format, its own conventions for instruction syntax, its own way of using memory, and so on. Once again, the strong emphasis on hardware efficiency led to the development of systems which, far from helping the people who used them, frequently seemed to make the tasks harder.

SOMETHING ELSE TO DO.

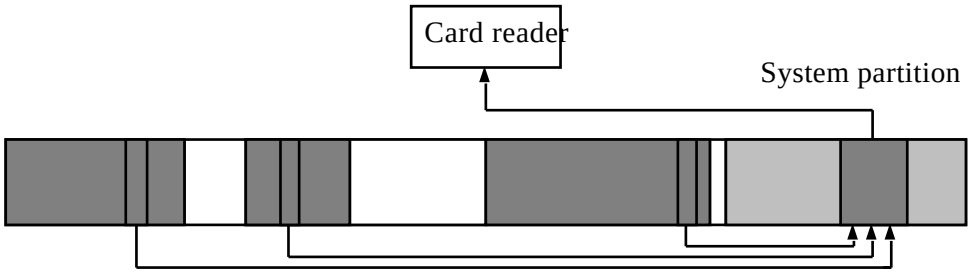
Ways of permitting several programmes to run "simultaneously" in a single computer were developed. Of all the changes in this collection, this is the most significant, and to some extent affects most of the others. One specific consequence was that the system now had to keep track of what was happening in many different places at once; so there developed many tables and lists of active programmes, memory usage, people using the system, active files, and so on. The monitor system only needed information about one programme, and typically spread it thinly through the code where it would be most useful; in the operating system it made much more sense to collect the information together.

CONTROL OF DEVICES.

People were no longer allowed direct access to any peripheral device; all input and output operations were mediated by procedures belonging to the operating system. If you don't exercise some sort of control, you have a system which looks something like this :



(The shaded areas represent the subroutines which drive the card reader.) In a shared system, this is obvious quite unmanageable. To ensure that the system remains usable, it is essential that some sort of system facility be provided to drive the card reader and make sure that it is only allocated to one process at a time :



Once again, though, the new feature forces us into yet further developments. We can rely on the new memory management hardware to preserve the system code from interference by other programmes, but the same hardware prevents any access to the system code at all ! We can get round this by providing a new form of machine instruction, known as a *system call* or *supervisor call*, which is rather like a subroutine call, and which can reach outside the local area defined by the memory management system, but which always jumps to the same location. The fixed destination is necessary, or the instruction could be used to circumvent the memory management; by predefining the destination of the jump, we can ensure that such calls are always caught by operating system code. A supervisor call is usually accompanied by some parameter (perhaps conveyed in a standard machine register) which identifies the action required from the operating system.

If you think a little about system calls and interrupts, you will notice that, except for their different origins, they are almost identical. In both cases, some event causes a transfer of control away from the current activity to some other code. For that reason, the mechanism is sometimes (as in MS-DOS and its derivatives) called a *software interrupt*.

This isn't the only way to run devices; the same effect can be achieved by trapping certain machine instructions – typically those intended to communicate directly with the devices – in the hardware, and executing corresponding system code procedures. This was taken to an extreme in the IBM "virtual machine" systems, where in normal running mode all input and output instructions were thus redirected. In effect, each trapped instruction becomes a separate interrupt, which comes to much the same thing as having a single interrupt accompanied by some parameter to select the required action. The difference is that, by trapping all device operations, it was made possible to compile or assemble programmes with the "real" machine instructions, just as you might do for the first of the diagrams above, but nevertheless to execute them according to the pattern shown in the second diagram.

A final requirement is that the actual hardware input and output machine instructions must be invalid unless executed by the operating system, so there are two *modes* in which the processor can operate, typically distinguished by such names as *normal mode* and *supervisor mode*. In normal mode, any attempt to execute a device instruction causes some sort of exception – either a fault or, with the virtual machine architecture, a branch to appropriate system code; in supervisor mode, the same instruction can be executed to control the device.

The result of this development was more than simply to stop two programmes using the printer at the same time. For the first time, there was a standard

mechanism for managing devices. In earlier days, each programme was responsible for driving the devices which were required to make it work, so if you wanted to use a strange device which you had built it was no harder than using a standard device – except that you'd have to write your own low-level software instead of finding it in a library. Under such circumstances, device management remained a jungle, and little progress in systematising and controlling it was possible. The use of supervisor calls introduced a simple mechanism which had to be followed by all programmes which used devices, and led the way to a sound systematic treatment of device management.

This was fairly clearly a good thing, as it brought a number of improvements in its train. An example is the development of *device independence*, which is the ability to defer the assignment of specific devices to be used until the programme is executed. In the earliest days, you decided when you designed a programme what devices, and perhaps even what file names, you would use. The programme was compiled with these definitions built in, and before running it you would change the name of the input file you wanted to use to match the name assumed by the programme. Before long it became possible to defer naming the file until the programme was running, but you still had to say beforehand that it was a disc file, so that the required subroutines could be linked into the programme. Once a standard way to handle all devices was developed, you could defer identifying the device until the programme was loaded, because (within sensible limits) all devices used the same interface; this is true device independence. Later still, we attained the current situation, where in many cases you don't need to think about what device you wish to use until you've finished making the file. This is an interesting progression of developments, and clearly illustrates how late binding is associated with increased flexibility in system use.

HIGHER LEVEL LANGUAGES for job control.

"Job Control Languages" were provided by the computer manufacturers. The best known was undoubtedly IBM's version, usually called simply JCL. Most were not easy to understand. No two were in any sense compatible. It was never really clear why not; perhaps part of the problem was that they were not perceived as "proper" programming languages. By this time, no one would have dreamed of writing a high-level conventional programming language to run on only one brand of computer – but the corresponding phenomenon was regarded as normal in the field of operations.

GENERALLY –

The effect of all these changes was to provide the ordinary programmer with a *virtual machine* (alternatively called an *abstract machine*). The programmer could no longer see the actual machine hardware, because the operating system was always in charge. The virtual machine was not necessarily the same as the actual machine : commonly, its memory size was different (initially sometimes smaller, but later on possibly much larger), access to devices could only be gained through system calls, and it contained no sign of any system code or other people's programmes. Peripheral devices which appeared to exist (as in IBM's interpretation of virtual machines) were not real, but somehow simulated by the system.

Apart from that, though, the change seen by the ordinary person using the computer was close to zero. The system was still a classical batch operating system : you still punched your operating instructions, programme, and data onto cards, handed the cards to the computer operator, and collected the cards and any printed output at some later time. All the changes were internal to the system; the aim was still very definitely to get the maximum possible amount of work from the computer hardware.

ABSTRACTION.

There's an important principle in this development which we shall find repeatedly throughout the study of operating systems – and, indeed, throughout the whole of computing. It is the notion of *abstraction*. The programmer sees a view of a virtual machine, which provides a certain set of facilities, and doesn't need to inquire into just

how these facilities are made to work. At a lower level, the operating system software has to support the facilities, but doesn't need to worry about how they'll be used. The abstraction of the virtual machine therefore makes it easier for everyone.

Alternatively, we can think of *clients* and *servers* : we say that the virtual machine provides certain services, which can be defined, to clients who can use them as they wish. Provided that the system maintains the service, and the clients use it according to the rules, everyone will be happy. This client-server model of the system at this level is little more than a mildly interesting alternative view of the relationship between different system component, but it has since become a very significant operating system design principle, particularly in the field of distributed systems.

The common feature which connects these two views is the interface. In both cases, we have to define the behaviour of (or the services provided by) the virtual machine; and, once the definition is made, each party need not worry about the details of what happens at the other side of the boundary. We can even completely rewrite the operating system – just so long as we ensure that the defined behaviour doesn't change.

- OR OBSTRUCTION ?

In our comments on abstraction above, we faithfully present the party line. In practice, things are not always quite so simple, because people believe the abstractions without, so to speak, reading the fine print. This happens at all levels; an early example turned up with the **sort** instruction included in the Cobol language. The **sort** instruction was a little more complicated than some others, but looked quite like instructions such as **add** and **move** and **perform** (a way of executing a procedure) and so on. Many people therefore took to using it just as they would **add** – so if they happened to want the largest element of an array, they would **sort** it and just take the first element of the result. (If you are not horrified by that statement, then your own understanding of sorting is inadequate. Go and find out.) Because the details were not visible in the interface, they were ignored.

A recent (1992) conversation with a data communications specialist reveals that nothing has changed. A large organisation operated an international computer network which relied on a set of routing tables to get its messages to the right places around the world. Over the years the network had become so complex that to build the tables from the raw information about the network could take up to 60 or 70 hours, which had become unacceptable. The specialist had been called in to consult. By taking account of what was really happening when the tables were constructed rather than the view presented by the database system used for the network information, he was able to produce software which reduced the time required to build the table to six or seven minutes.

If you want a moral from these stories, try this : *abstraction is an aid to, not a substitute for, thinking about complex systems.* Intelligent use of abstraction can help you to solve complex problems – but it's still your responsibility to make sure that the abstractions are being used sensibly.

QUESTIONS.

What facilities do you need to implement adequate security ? Can it be done with software alone ?

How can you prevent people from using peripherals without going through the operating system ?

What facilities do you need for multiprogramming ? What part of the system must perform the jobs required ? (Do NOT be satisfied with answers like "it will direct the interrupts to different parts of memory". WHAT will direct the interrupts to different parts of memory ?)

Given that an operating system implements a virtual machine, are there any limits on the degree of virtualness which makes sense ?
