DOING TWO – OR MORE – THINGS AT ONCE

Once it became clear that the road to greater efficiency led through multiprogramming, all that remained was to find out how to do it. There are really two questions which must be answered :

- How can we fit two (or more) programmes into memory at the same time ?
- How can we keep track of where all the programmes are up to in their execution ?

We shall defer the answer to the first question to the next chapter.

INTERRUPTS.

A partial answer to the second question was provided by the invention of *interrupts*. An interrupt is a hardware operation which we don't propose to discuss in great detail – if you want to know more about interrupts, refer to texts on computer hardware techniques. Briefly, though, a processor's interrupt machinery provides a way to switch the processor immediately from its current task to something else in response to an electrical signal; it is commonly used to make the processor give attention to some event which isn't under its control. Suppose a programme is running. An interrupt saves the address currently in the processor's instruction address register (programme counter) somewhere, and replaces it by some other address which is directly or indirectly under programme control. If the code executed at the new address makes appropriate provision to store the state of the processor, including the saved old address, then the original programme can later be resumed as if nothing has happened. If the interrupt branch address is in the code of a second programme, the result is to switch from one programme to another; we have achieved multiprogramming of a sort.

It's a useful sort, too, but unfortunately it isn't quite the sort we want. It's useful, because it does give us the ability in principle to jump from one programme to another, but it falls short of perfection in that it isn't much use if the programme you want to run, like most ordinary programmes, doesn't have an interrupt attached to it in some way. In effect, the multiprogramming doesn't go far enough : in order to achieve the target of always having a programme ready to use an idle resource, we must run several jobs simultaneously. That's why we described interrupts as a *partial* answer to the question of how to keep track of all the running programmes. There are ways and means of achieving the desired end, and we'll come to these later, but these interlocking problems suggest that we are going to need a new approach if we are to advance much further.

If we do have the right sort of interrupts, though, they are very useful to control switching between activities, and it's worth describing an example which started many years ago but which is still useful as an operating system component.

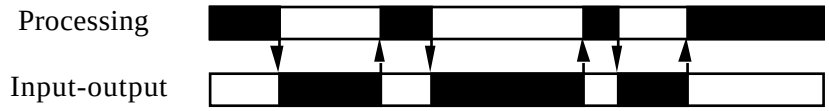
SPOOL.

From where do the interrupts come ? The usual source of interrupts is peripheral devices, so the obvious application of the method described is to driving some peripheral device with the interrupt-handling programme while carrying on some memory-intensive computation in the "original" programme. The arrangement is sometimes described as a computational job running in the *foreground*, with a programme handling a peripheral device in the *background*; be warned, though, that the terms "foreground" and "background" relate entirely to people's perception of the relative importance of the jobs, and are not synonyms for interrupted and interrupting, respectively.

Just to show that this confusion isn't new, here's a snippet from the past[HIS3] :

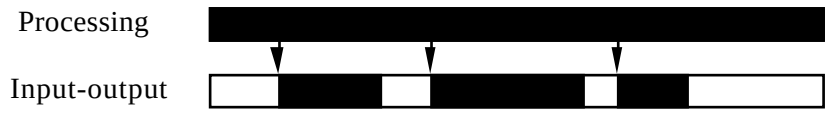
> "***foreground processing*** *1. In a* multi-access system*, processing which is making use of* on-line *facilities. 2. High priority processing which takes precedence (as a result of* interrupts *) over* background processing*. 3. Low priority processing over which background processing takes precedence.*
>
> *"As definitions 2 and 3 are directly contradictory and definition 1 has a related but different meaning, this phrase should be used with caution."*

Some diagrams might help to illustrate the idea. Here's a representation of the operation of a system without interrupts. At any moment, either the processor is active (represented by the black trace) while the input-output system is inactive (white trace), or vice versa. The meaning of "inactive" is not defined, but whatever it is it doesn't contribute to the task being performed. The inactive processor might be continually polling the input-output device to see whether it has finished, or it might be directly involved in driving the device.

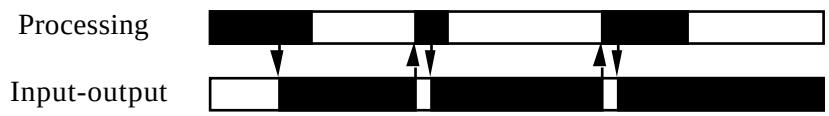


It might sometimes be possible for the processor to do some work in the inactive period, but details depend on the level of attention needed by the device.

Once interrupts are possible, there is a mechanism with which the device can get attention from the processor at any time, so much less caution is needed. Now the processor can be occupied on useful work – if there's any to do, which depends on the nature of the programme. These two diagrams illustrate the possibilities.



Overlapping : processing predominates



Overlapping : input-output predominates

An early, and very successful, application of this idea was in the SPOOL system, originally introduced by (we think) IBM. This was a way of achieving the benefits of off-lining (see the chapter *EFFICIENCY : ...*) without investing in several computers. The useful work was carried on by a main programme, and two interrupt-driven service

programmes ran permanently in the computer to copy input card decks to the disc, and output disc files to the printer. This was called **S**imultaneous **P**eripheral **O**peration **O**n **L**ine – but not many people remember that, and "spool" is now more commonly used as a verb or adjective.

It is worth pointing out that the spool principle requires very little, if any, intervention by the system software. So long as there is a way of starting the service programmes at the beginning of the operation and setting the required interrupt branch addresses, and of changing the main programme when necessary without interfering with the service code, the system can run, and can keep on running. The system code isn't involved in the transfers of control between the programmes; the interrupt causes the branch by hardware, and the interrupt routines look after saving and restoring the contents of such registers as must be preserved.

The main constraints on the system come, once again, through the definition of conventions, particularly in the matter of memory addresses. If the service programmes are there all the time, then the main programmes must be linked and loaded to use only what's left of memory, and it is obviously sensible to build the correct addresses into the system software, at least as default values. Programmers will also need some way to call on the services of the service programmes; an easy way to provide this facility is to make the various services provided look like subroutines, so that the programmers can call them like any other subroutine. We're still calling these system conventions, but we can see the beginnings of what we'd now call an *Application Programmer Interface* (API).

The main defects of the spool, and similar, systems are that the device-handling programmes are vulnerable to overwriting by misbehaving main programmes, and we can do little about that without hardware help in the memory management area, which brings us back to the first of our two questions at the beginning of the chapter. Despite that, they provided good service for quite a long time, and systems of this type are still generally used for jobs like printing.

REFERENCE.

HIS3 : A. Chandor, J. Graham, R. Williamson : *A dictionary of computers* (Penguin, 1970), page 167.

QUESTIONS.

We claim there is "no simple extension of Spooling". Can you design one ? If not, why not ?
