SCHEDULING – IMPLICATIONS.

We defined scheduling as determining who can do what, and when. The simplest solution is anarchy : anyone can do anything at any time. Whatever the merits of anarchy as a political philosophy, it has flaws if considered as the basis for an operating system scheduler. On the other hand, if we go back further to our description of the aims of an operating system as to provide computing services to people who want them, anarchy looks like the perfect solution.

As is common, then, we are seeking a compromise. We would like to achieve the closest approach to anarchy which is compatible with running an effective computer system. This directs our attention to the factors which hinder anarchy – which is to say, finite resources. To demonstrate the correctness of that conclusion, put yourself in the position of a process running in a computer for the moment, and ask yourself why you wouldn't be able to do anything that a process might want to do. The only answer is that you can't get the resources you need to do it.

Assume that there is no reason in principle why you shouldn't be able to proceed with your task. Then either the resources exist, or they don't. If the resources don't exist (perhaps they are messages or signals of some sort), then you have to wait until some other process produces them for you. If they do exist, then some other process is using them, and you have to wait for it to finish. We've been through this sort of argument before (in *WHEN SCHEDULING FAILS*), and won't repeat it, but the general conclusion is that the delay is either because of competition between processes for physical resources, or because of the need for communication between processes.

Well, then, we can solve all our problems by giving everybody enough resources (whatever that means) and not trying to write programmes which need communicating processes. That's called giving everybody a computer, and it is indeed a complete solution – for the people for whom "enough resources" isn't prohibitively expensive, and who never want to write programmes which need communicating processes. That's why our laboratory works so much better than the various sorts of batch and timesharing systems we used to use to provide services to students. (And, yes, some of you would doubtless like a 100 GHz processor each, with 128-bit data paths, and a few A0 screens with 1000 pixels per millimetre and infinite colour palettes, but the machinery in the laboratory is about the right size for the tasks we expect you to accomplish on it.)

That sort of solution is becoming more practicable as the cost of hardware drops, and microcomputers become more and more powerful. But they are already multiprogramming, which necessarily gives us competing processes, and moves us closer to communicating processes. The arrival of multiprocessor microcomputers will further complicate the systems (which is why we didn't offer you a lot of 100 GHz processors each); so it looks as though the solution to the problem simply brings the problem back again ! What else can we do ?

If we can't get rid of the competition and communication, we shall have to be rather more subtle in our approach. We might be stuck with scheduling at all the various levels we've mentioned for ever, but perhaps we can be clever about it. To do so, we consider the nature of scheduling itself : it is wholly a matter of *making decisions.* Given a fairly free interpretation, we can say that the purpose of every sort of scheduler is continuously to answer the question "What should happen next ?". A primitive scheduler can answer the question by organising all requests into queues and always answering the next one in turn; but we have seen that we can do better than that provided that we have more *information* on which we can base decisions.

We can make use of an astonishingly wide range of information at different levels of the scheduling system. Here's a list, which is certainly incomplete, but illustrates the point : system configuration; characteristics of parts of the system; historical performance; overall workload patterns; current system performance; expected resource requirements; current behaviour of individual processes. It's an extensive list, but it is *completely useless* unless the operating system has access to all the information

whenever it is needed. This is perhaps the major implication of scheduling : a good scheduler must be supported by a lot of information, which must be correct and up-to-date.

How do we get the information ? There are three sources, corresponding to rather different ways in which the information is used in the system.

Configuration information is supplied when the system is started. It includes details of what devices are available, how the disc areas are allocated, and other general information about the system. It is quite possible for information of this sort to be used when the system is being set up, but then lost when its initial purpose is over – so the system might set up its devices satisfactorily, but then forget what devices it has unless the information is saved in something like a device table.

Running information is used while the system is in operation; most of the system tables come into this category. This material at least has to be present and current, but that doesn't necessarily make it easily available outside the parts of the system which normally use it. Most of the time, that doesn't matter, but it's a nuisance if you need it and can't find it.

Performance information might not be used by the system at all, but is very important in planning for developments. If this sort of information is not collected and stored, it will be lost. Notice that this can reasonably include at least some of the information of the other two categories, much of which is important in identifying the resources available and the system's behaviour.

The newer systems are much better at collecting information and making it available than older implementations, but performance is still scrappy in the smaller systems.

QUESTIONS

Consider our laboratory. It isn't designed to let you do everything you want to do just when the fancy takes you. Why not ? How does it fit into the scheduling pattern ?

How can each of the sorts of information in the list above be used for scheduling ? (Some can be used in several ways.) Can you extend the list ?

One effective scheduling technique with batch systems was to record statistics on the behaviour of jobs, all of which had names, and then to assume that the requirements of a job would be the same as they were when it was run last time. Why did that work ? Would it work with a timesharing system ?
