SCHEDULING IN ACTION

So far in this section we have mentioned seven aims of scheduling, five levels, and at least four operating modes. Here we try to point out how policies and actions implemented at four of the different levels can assist in attaining the overall aims. We've omitted the interrupt level, because it allows us very little room for manœuvre. The single relevant principle at this level is : KEEP ALL INTERRUPT PROCEDURES AS SHORT AS POSSIBLE. Generally, if an interrupt signals that a lot of work is to be done, the interrupt handler should delegate the work to a conventional process, and spend as little time as possible setting up the process before returning from the interrupt.

There is a fundamental problem with scheduling which in principle makes the whole idea impossible : every action we take to improve the system's performance is based on the assumption that we can foretell the future, so we know what the consequences of the action will be. In practice, it is only in rather specialised cases that we know what a programme is supposed to do, let alone what it actually *will* do. Fortunately, we can often make a reasonably good guess at what will happen; we can guess what resources a programme will need when executed from our knowledge of what it's supposed to do this time and its previous performance, and on a smaller time scale we can assume that a process will do much the same thing next time slice as it did in the previous one, but there is always a degree of uncertainty.

In many of the cases we discuss below, there are several answers, depending on just how the computer is – or the computers are – being used. The requirements of batch and interactive work are different, and the performance of systems operating these different regimes is optimised in different ways. Further, it is not necessarily the case that everything happens inside one computer. What if instead you have lots of computers ? You could, for example, have a laboratory full of isolated microcomputers, or a distributed system with a lot of computers connected by a network. What sort of scheduling do you need then ? This question is important in several cases, some of which we shall mention in the short review which follows. If the answers don't always sound much like scheduling, recall that we defined it as deciding who can do what when.

Here are some remarks on some points of interest in some of the possible combinations of circumstances found in practice. We have attempted to classify the comments according to three sorts of system characteristic which we have already mentioned in the chapter *SCHEDULING : WHO CAN DO WHAT, AND WHEN* : we called them the system's aims, levels, and modes. Despite this attempt at breadth, we should warn that our treatment is not intended as an exhaustive discussion. Consider it rather as an indication of the variety of techniques which can be used to improve performance, and also as a reminder of the complexity of the system to be improved. Notice as you read that many of the policies are double-edged; while they help to achieve one of the aims, they might hinder others. Perhaps the best known example is the conflict between keeping machinery in use and providing instant service in an interactive system. Because of these complex interactions, any large operating system is a collection of compromises which we hope will result in acceptable performance under all relevant headings.

AIM : KEEP THE MACHINERY GOING.

If something isn't being used, try to use it. With the current comparatively low hardware costs, this aim is much less important than it used to be, but it is still sensible to do work when you can.

Administration : Aim for a balanced configuration – make sure that you have enough memory, disc channels, etc. to serve the processing load you want. Statistics of the numbers and types of job which have been run in the past are valuable in determining good values for the various system parameters. (Batch) Set characteristics of job queues to facilitate the actions of the long and low-level schedulers. (Interactive) Make sure that you have enough memory, disc channels, network capacity, etc. to serve the *peak* load reasonably well; look for ways to persuade people to move their work from peak to off-peak times. (Isolated) Select booking and usage policies to ensure that idle machines can always be used if there is work which they could do.

High-level : (Batch) Choose jobs to match the demand to the load. All the automatic decisions depend on knowing what's coming; there's no reliable way of finding that out by inspecting the submitted jobs, so we have to rely on the jobs' owners telling us. That's accepted for batch systems, but deemed impracticable for interactive systems – reasonably enough, for you always know what's involved in a batch job while an interactive session can develop in ways you didn't really expect when you began. (Distributed) Move work onto devices which are not being used; optimise file and processing sites to reduce comparatively slow inter-computer operation depending on network traffic.

Low-level : (Batch) Schedule processes to match the demand to the load. (Distributed) Use the network facilities to balance the load over all available processors.

Dispatcher : Can't do very much directly.

In some cases, it might be impossible to achieve this aim in practice. Even if some facility of a system isn't being used much, it could nevertheless be necessary to keep it if it is essential to those who *do* use it. An administrative stratagem which can help to maintain the principle of keeping things going, and incidentally increase the cash flow, in such circumstances is to make it available to a wider community through networking. By this means, facilities such as databases or specialised hardware (such as supercomputers) are generally available through the internet to anyone who needs them and can pay whatever fees might be required.

It is also possible that attempts to make use of idle equipment can turn out to be counterproductive. A classic example is the attempt to combine batch and interactive systems. A major difference between batch and interactive systems follows from the lack of urgency in the batch system. Unlike the interactive system, it can plan ahead and attempt to optimise the flow of work through the system by carefully scheduling jobs, taking into account their different characteristics. An interactive system, in contrast, must respond instantly to instructions from those using it, so cannot plan ahead; in consequence, it must underuse its resources most of the time if it is to be able to cope with sudden changes in demand without serious degradation. In a mixed system, these strategies conflict. Unless measures to the contrary are taken, the system cannot benefit from the characteristics of the batch stream because its careful scheduling might at any time be upset by unexpected demands from the interactive work; while the resources available to the interactive stream are limited by the demands of the batch work.

AIM : KEEP THE SPEED UP.

Administration : Don't overload the system. Set limits on the number of things going on at once, by (batch) setting parameters in the high-level scheduler or (interactive) not trying to run too many terminals. (That doesn't work as well as it used to, now that many operating systems are quite happy for people to run multiple processes in one way or another, but in practice most people stick to fairly straightforward computation so acute problems rarely arise.) (Network) Cut down time delays for network services; many devices (such as printers), network spooling, faster network.

High-level : (Batch) The high-level scheduler controls the overall traffic flow through the system. It must monitor parameters of jobs as they are presented, and only accept work which can reasonably be run soon. Minor errors of judgment are rarely very significant, as processes are still under fine control of the low-level scheduler, but a job once accepted begins to take up space in the system queues and time for the low-level scheduler.

Low-level : (Batch) Suspend, or don't reschedule, processes if there are signs of overloading.

Dispatcher : The dispatcher might run many times in every second, so the most useful thing it can do is very little. When designing an operating system, it's possible to think of all sorts of useful and interesting things that could be put into the dispatcher, but on analysis very few of them return enough benefit to justify the time spent on the computation.

The major technique is to attempt less so that you can do more; the aim is to avoid congestion and interference between different processes and the system. Of course, if too little is going on, we aren't keeping the machine going, and resources are wasted. There is evidently an optimum workload, but it is very hard to determine.

In a multiprocessor system, you have an additional degree of freedom : if you have spare processors, you can give anyone more than one if it's useful to do so. This works both with tightly and loosely coupled systems, though the programme organisations appropriate are different in the two cases.

AIM : AVOID UNNECESSARY WAITING.

Administration : Tell people what's going on, so that they can schedule their own work sensibly. (For example, give long lead times for students' assignments so that they can schedule their work to avoid congested periods.) Publicise maintenance times and breakdowns, and the system load throughout the day. (No, we know you don't schedule breakdowns, but if anything serious does go wrong you can try to inform people as soon as possible.) Keep track of performance, acquire more, or better, machinery when loads become heavy. If your system is very heavily laden, you can use differential charging to encourage, or some sort of privilege structure to force, people to work at unpopular times. Priority structures associated with different classes of work can control who has to wait when there's a conflict; just whose waiting is more unnecessary than whose is a management decision.

High-level : (Batch) Try not to accept jobs known to rely on heavily laden resources – not always easy, because many systems run a lot of jobs which are automatically scheduled, where it doesn't make much sense to say "Not now, come back later". (Interactive) Make sure that the system gives sensible responses to requests for unavailable resources – ask whether the process should be stopped or queued. (Isolated) The booking system again. (Well, it's all we have.) Discourage "just in case" bookings which might never be taken up, but which force other people to make later bookings than they had wanted.

Low-level : There are ways to ensure that tasks of low priority don't wait for too long. A common method is to increase the priority of every waiting process from time to time, so that even a low priority process will eventually gain enough priority to be executed.

Dispatcher : Use a policy which guarantees to avoid starvation; policies which take a process from every queue once every N cycles, where N might be different for different queues but is always finite, are better than policies which only inspect certain queues when other queues are empty. (But recall our earlier remarks on clever dispatchers.)

In a single-processor multiprogrammed system, a lot of work is always waiting. The trick is to identify *unnecessary* waiting, and do something about it. Unnecessary waiting is hard to define; we've identified it here as any waiting for a resource which either the operating system or the person using the system knows, or could know, to be unavailable, or as waiting longer than necessary because of misleading information, or as starvation.

There might in addition be degrees of necessity, as suggested in the note on Administration. Some of this is inevitably associated with the idea of job classes – if you have a class intended for short, fast jobs, it is reasonable enough to give jobs in this class priority.

AIM : BE SEEN TO BE FAIR.

Administration : Be open. Provide information on how the system runs, who can do what, and when, and so on. Explain any temporary changes, and what their effects will be. Do not impose policies which you can't enforce, or which can be circumvented by people with intimate knowledge of the system, or low cunning. (Everything but batch) If you have too few terminals or computers for the people who want to use them, then you have to institute some scheme for rationing and booking. This is what we do in our laboratories. What we want to do is make sure that everyone who needs computer services for our courses has access to whatever is required, and we try to do it by providing a booking system. It's far from perfect. What we should also try to do is make sure that people can't use too much time, thereby keeping others away, and that they only do work connected with their courses in the Computer Science department – but that's harder.

High-level, Low-level : Nothing very specific. Queues should be ordinary queues throughout, not bypassed except in justifiable and well defined circumstances. The important thing is to implement the rules as publicised – so if you claim that, say, intensive processing jobs will be penalised, make sure that it happens, and that there is no loophole. Priorities, if any, should be seen to be observed – but note the earlier remarks on making sure that even low priority processes will be run sometime.

Dispatcher : Implement a priority mechanism (for example, using a multiple-queue dispatcher) which can penalise processes which don't fit in with the system's aims. For example, a process which is slowing down interactive response by using the processor intensively can be slowed – or, in a batch system, encouraged !

Fairness is a system-wide principle, much of which has to be implemented outside the schedulers. It starts with administrative decisions about how the system will be run, but these have to be reflected in the software. Experience suggests that most people will accept reasonable policies, given reasonable explanations, but that – sadly – some won't let that cramp their style if they can find ways of getting round restrictions. It is therefore unwise to promulgate rules which you can't enforce in the system hardware or software.

Two examples, from the history of this university :

With the reasonable enough intention of keeping as many people happy as possible, and of clearing as many bodies as possible out of the immediate vicinity of the computer system, the Tops-10 operating system printer software would choose the shortest file from its queue to print first. This was very pleasant if your file was short. If, on the other hand, your file was 100 lines long (normally considered a short file) but you tried to print it on the day before the deadline for a large programming class's assignment requiring a programme listing of about 50 lines, a simple first-come, first-served queueing discipline suddenly became very attractive. (Or, of course, you could spend a little time splitting up your long file into several short ones, which made nonsense of the intention of the queueing discipline and in fact created yet more congestion.)

When terminals for the IBM4341 machine first began to appear around the university in buildings other than the Computer Centre, they were typically collected in clusters of several terminals and a *printer. Early experience showed that the printers associated with the clusters, which were reliable but rather slow, could very easily build up queues of work which would take hours to print. It was therefore decided that file of more than 1000 lines were not to be printed on the remote printers, a rule to that effect was promulgated, and the locally written command file used for printing files amended to check the file length and take appropriate action. Unfortunately, it was impossible to hide the operating system's own printing software, so anyone willing to put in the additional effort needed to use that could circumvent the regulation quite easily.*

AIM : KEEP EVERYTHING MOVING.

Administration, High-level, Low-level, Dispatcher : Nothing special – covered by earlier requirements.

An important extension of this principle is to make sure that people using the system can tell that things are happening; a batch system can display the current system state, and an interactive system can provide ways to find out what each currently active programme is doing. (Batch) Keep the *system* going; it doesn't matter if individual processes are suspended for a minute or two. (Interactive) Keep *every process* going – that's much harder. (Systems involving people) Pay special attention to popular times for moving about – particularly hours and half-hours, and ends of bookable periods. At these times there is a sudden increase in demand for whatever facilities are used for logging in and out, booking, and printing.

AIM : BE RELIABLE.

Administration, High-level, Low-level, Dispatcher : Nothing special – everything must be designed to cope satisfactorily with overload, and as well as possible with breakdown.

Again, not restricted to the schedulers. It is essential that people know that if they submit a job to the system then either it will be completed as expected, or that they will be told what happened. (Silly jokey boxes showing pictures of bombs or beetles and messages saying "System error" are symptoms of incompetence; when accompanied by buttons labelled "Resume" which you can rarely, if ever, use, they indicate a degree of arrogance and contempt for "users" which should be actionable.) If people don't have confidence in the system, they are likely to try again if it doesn't respond as they think it should – which, in an interactive system which is merely running slowly rather than faultily, might make congestion even worse. Notice that this brings us back to the "system genie" again; if people have a clear model of the system, and it always works, then they're more likely to be patient if it sometimes runs slowly. (Distributed) Make sure that the network is as reliable as the computers which it connects. (Or, in some cases, vice versa.) (Systems involving people) Don't forget that the booking system has to be reliable too.

AIM : AVOID SUDDEN CHANGES.

Administration, High-level, Low-level, Dispatcher : Nothing special.

Again, not restricted to the schedulers. Typically, the sudden changes happen when a process suddenly begins to make large demands for resources. (Batch) If you know that the process might need a lot of resource, then make sure there's plenty available before you schedule it. More commonly, though, you can't predict the request, and when it happens the process is (obviously) already running.

SCHEDULING AND PEOPLE.

If you look back at our classification of the seven scheduling aims (in *KEEPING THE SYSTEM RUNNING SMOOTHLY*) and compare it with the remarks above, you will observe that we have given fairly plausible suggestions for the three aims concerned with maximising the work done, but have done much less well in recommending how to keep the people happy. All we can say to that is that no one else knows how to do it either. It is not too hard to satisfy both requirements reasonably well with a batch system, where people do not expect instant service, but with any sort of interactive system it seems to be impossible. The best approach seems to be to spend a very large amount of money so that everyone can have a very big, very fast machine. This policy runs counter to the unstated, but understood, aim of not going bankrupt.

There are two reasons for this difficulty, both firmly rooted in the people side. First, many people seem always to want just a little more than they have, so live in a chronic state of dissatisfaction. Short of prescribing a course in basic philosophy, we can't do much about that. Second, although we said that in interactive systems there was no equivalent to a low-level scheduler, in practice there is indeed a low-level scheduler, but it lives in the person rather than the computer, and is therefore unfortunately outside our control. It is this scheduler which can severely reduce the work done by a computer system by scheduling thinking time or coffee time or chatting time or dreaming time or (in extreme cases) sleeping time. Such activities are a nuisance in a shared system; they can be a serious drain on a system's resources, and are not uncommon when people have to book time in fixed slots but run out of things to do before the end of the booked period. Thinking time is fair enough if the broader system including the person as well is doing something productive, but it's impossible to distinguish from the others automatically, so we can't compensate for bad scheduling at this level elsewhere in the system.

QUESTIONS.

We remarked on the clash between batch and interactive operation when the two are combined. In designing such a combined system, it is proposed to reserve a certain area of memory for the sole use of interactive processes, and allow them access on request to the disc and processor, with all other equipment in the system – other than terminals – accessible only from batch processes. Would that help ? Why (not) ?

How would you expect the performance of a single combined (batch and interactive) system to compare with that of two identical computers, together equivalent in computing power to the single computer, with one computer reserved wholly for batch work and one for interactive work. ?

How could you improve our laboratory booking system to make it easier for everyone to get a fair share of the facilities ?

In a university computing laboratory, students are allocated a certain amount of time for each course which they are taking. It is observed that there are occasions when students who would like to use the computer for legitimate educational purposes, but have used up all their time, have to remain idle while there are terminals (or microprocessors, or whatever) also lying idle. What sort of scheduling system can you devise which would allow the impoverished students to use the computer in slack periods ?
