Computer Science 773

Robotics and Real-time Control

SCHEDULING TASKS

Scheduling computing tasks in real-time systems is difficult but necessary. It has always been recognised as a problem, but the effort to find solutions is complicated by constraints of safety : if the solution is not perfectly safe, the consequences might be much more serious than in conventional computing.

A primary difficulty is in knowing how much resource the processes to be scheduled will require, and, in particular, how long they will take. This is perhaps the only respect in which scheduling for real-time systems is easier than it is for operating systems, because it is usually possible to assume that the set of processes which must be used is completely known, and that the processing requirements of each process can be determined beforehand. This is not so in a general computing system.

LEINBAUGH'S METHOD : WORST-CASE ANALYSIS.

This additional knowledge is used in very early method of scheduling described by Leinbaugh. It's very much a brute force method which works by attempting to enumerate all cases, but is still a step ahead of the "try-it-and-see" approach more commonly used. It is notable for the attention given to all the processes' requirements, and the attempt to deal with the worst-case. That argument is that a schedule which is known to handle the worst case will certainly handle anything else, so is guaranteed to satisfy any feasible constraints.

The type of reasoning used by Leinbaugh is illustrated in this simple example. Suppose that we have a system in which six processes must be executed every second. The processes themselves cannot be interrupted once started, and only one process can run at any moment. Five of the processes, S1 to S5, take 0.1 seconds each to run and must be run in order, while the other, T, takes 0.4 seconds to run, but must await an interrupt (guaranteed to arrive in the first half-second) before it can start. Given these conditions, we cannot state just when within the one-second period any one of the processes will be executed, but we nevertheless can guarantee that all processes will be completed within the period.

Leinbaugh extends that idea to cover all possible cases which might arise in process control. He gives an exhaustive analysis of all the factors which might cause a process to run or not run, and requires a lot of information about the processes themselves. I don't know of any study of the mathematical complexity of his method, but it looks like a combinatorial nightmare. Nevertheless, work continues; Leinbaugh extended his original work to the case of distributed systems, and more recently (1997) further work on finding more efficient algorithms has been reported.

LESS PRECISE METHODS.

Leinbaugh's method is precise, but expensive; we can develop alternative methods which are less expensive at the cost of lower precision. Given the basic timing information, it is possible in principle to compute schedules of various sorts for given time constraints (or to demonstrate that no such schedule exists).What sort of solution do we want ? Here are some possibilities :

- **Optimal scheduling :** The information is sufficient to compute the best possible schedule for the constraints. Unfortunately, to do so is time-consuming; the

problem is known to be NP-complete, and to determine an optimal schedule for a large system is prohibitively expensive in time.

- **Feasible scheduling :** Provided that it can be done at all, it is usually much easier to find some schedule which satisfies a set of constraints. The difficulty in this case is to make sure that the schedule will still work if the actual times of events vary a bit; if just one event happens slightly late, you can get what amounts to a delay amplification behaviour, which might ruin the overall process.

- **Guaranteed scheduling :** What we really want is a schedule which we know will do the job. It doesn't have to be optimal, provided that we can afford to run it; it does have to cope with predictable ranges of variation in timing and such parameters. To get such a schedule, we might have to make conservative assumptions, but it is generally thought to be worth it.

In practice, all the useful methods rely on some sort of task classification, where it's possible. If we can find different sorts of difference between tasks, we can construct scheduling methods which use these differences as a way to improve scheduling performance. Here is a selection.

TASKS OF DIFFERENT FREQUENCY.

The easiest way to guarantee that your computer will keep control of the plant is to make sure that the plant can't mess up the activities of the computer. Essentially, that means giving up interrupts from the plant, so all communication must be initiated by the computer. To make sure that everything is dealt with sufficiently promptly, the computer must therefore poll the plant for changes in its condition at "sufficiently frequent" intervals. This gives rise to the *cyclic executive* scheduling method, where the controller consists of a sequence of polling operations covering all the interesting plant variables, and repeated cyclically in sequence for ever. Action is taken when the polled values indicate that it's necessary.

In practice, not everything need be polled al the time. Some variables change much more slowly than others, or need much less precise control. It is therefore sensible to sort out the processes according to their frequency and operate several polling cycles at different speeds. One speaks of major and minor polling cycles.

This works well, but is very expensive in processor time. If an event must be acted upon within 0.01 seconds, then it must be polled at least 100 times per second – even if it only occurs once every hour. The cyclic executive works, but there's a significant cost to its reliability.

TASKS OF DIFFERENT URGENCY.

A different approach is to rank processes in terms of their *priorities*, and to take the priorities into account when scheduling. This is a common scheduling technique in operating systems, and it has been used in some real-time controllers. It has one serious deficiency : using a system based on priority alone, it is very difficult to guarantee that the required level of service can be met, because another task of higher priority might intervene at a critical time in the performance of a task of lower priority.

One can argue that the idea of priority doesn't really fit real-time systems. The existence of deadlines means that you can't really define a fixed priority, because that doesn't take the deadline into account. In operating systems, a common device is to increase the priority of a queued process according the length of time for which it has

been queued; without a lot of not-entirely-trivial calculation, that just isn't good enough for real-time use. (In an operating system all we need is a guarantee that a process will leave the queue at *some* time, but we're not much concerned just when it happens.) The big difference is that in a real-time system you can't simply allocate a priority to a process alone; its urgency depends on its own deadline, and on the deadlines of other processes. Once again, the effect of the real-time constraints is to force an unwanted, but inevitable, degree of interdependence on the processes.

In effect, by the time you've done enough calculation to work out something like a priority which would work with a real-time system, you've done enough to complete the scheduling by something like a worst-case analysis, so the priority figure itself isn't particularly useful. Alternatively, you could regard the other scheduling methods a devices to set and implement a sort of priority system.

There is an exception to this conclusion. The argument above applies to the problem of working out a fixed task priority by considering the properties and resource requirements of the task in the context of the system as a whole, and as part of a scheduling exercise undertaken before the system is running. It does not take into account the events which occur during the run, which a short-term scheduler might have to take into account. For example, variations in conditions might in some cases delay the initiation of a task to the extent that it has only just enough time to run before its deadline; this is certainly a sort of urgency which a scheduler can take into account while the system is running. Remember, though, that it is very important that a short-term scheduler (otherwise called dispatcher) should not be overburdened with computation, so there is a limit to how much testing and checking it can do. Comparing the current time with the latest possible starting time for a process is sufficient to identify the urgency in the case described, and quick enough to be acceptable, but more elaborate calculations might well be too time-consuming.

TASKS OF DIFFERENT NECESSITY.

An interesting classification is based on whether or not you really need to execute the processes to be scheduled. This has some variants; here's a list, taken largely from examples in an issue of *Operating Systems Review* on real-time operating systems.

Inessential processes : Some tasks simply don't need to be done. If you haven't time to display the outside temperature on the screen, it might not matter. Such tasks can be dropped if time is short. This distinction is used in the Spring Kernel.

Not always essential processes : If you miss out polling a value once every couple of hundred times, it might not be important. While it's better to maintain values up to date when possible, many control systems work pretty well provided that their data are about right. They'll work better with the right data, but missing one or two points now and then won't destroy the system. A variant in which each task has a mandatory and optional part has been investigated.

Alternative processes : Sometimes you can get an approximate, but adequate, value faster than a precise value. For example, you might be able to estimate a reading from internal data faster than you can read the real value from a sensor, or you can provide a low-resolution display instead of a better high-resolution version. If there's enough time, you do the slow and better routine, but if not take the short cut. The Chaos and Spring Kernel systems use this distinction in their scheduling algorithms.

Soft deadlines : Even in a hard real-time control system, not all deadlines are necessarily hard. By distinguishing between tasks on this basis, a scheduler can choose a task with a hard deadline in preference to a softer alternative in cases where time is short. The Spring Kernel uses this approach too.

INFORMATION.

The common feature of all these methods is their reliance on having lots of information about the tasks which are to be scheduled. You can do that with process control systems, because the process is well defined, and – all being well – there are no surprises. in contrast, an ordinary operating system must cope with any processes which happen to turn up, whatever they are, and cannot be designed with detailed knowledge of what it must handle.

REFERENCES.

Leinbaugh's scheduling : D.W. Leinbaugh : "Guaranteed response times in a hard real-time environment", *IEEE Transactions on Software Engineering* **6**, 85-91 (1980), as reprinted in R.L. Glass : *Real-time Software* (Prentice-Hall, 1983)

Further work on worst-case methods (not very informative) : http://www-iiit.etec.uni-karlsruhe.de/~bort/.

The Spring Kernel : J.A. Stankovich, K. Ramamritham : "The Spring Kernel : a new paradigm for operating systems", *Operating Systems Review* **23#3**, 54-71 (July, 1989).

Mandatory and optional parts : W.-K. Shih, J.W.S. Liu, J.-Y. Chung, D.W. Gillies : "Scheduling tasks with ready times and deadlines to minimise average error", *Operating Systems Review* **23#3**, 14-28 (July, 1989).

Chaos : P. Gopinath, K. Schwan : "CHAOS : why one cannot have only an operating system for real-time applications", *Operating Systems Review* **23#3**, 106-125 (July, 1989).

Alan Creak,
March, 1998.