HANDLING DIFFICULT FAULTS IN OPERATING SYSTEMS

R.M. Needham

Computer Laboratory
University of Cambridge
Cambridge, England

It is commonplace to build facilities into operating systems to handle faults which occur in user-level programs. These facilities are often inadequate for their task; some faults or incidents are regarded as so bad that the user cannot be allowed to act on them and this makes it difficult or impossible to write subsystems which give proper diagnostics in all cases, or which are adequately secure, or which are adequately robust. This paper looks into why there is a need for very complete facilities and why there is a problem about providing them, and proposes an outline structure which could be used.

A prime requirement is for all occurrences which happen during the life of a process and which affect it to be capable of being acted upon within the process. This is clearly in some conflict with the overriding need that programs function solely under the jurisdiction of the operating system or at any rate of some superior, and that this superior may arbitrarily terminate or abolish them. Since such action must be provoked by something, one has on the face of it introduced a class of incident with which programs are not allowed to cope internally. Various devices may be used to mitigate the effects of this contradiction, but, rather than discussing them at once, it is preferable to consider why the initial principle was set up at all.

The basic reasons are diagnostics, security, and robustness. Taking diagnostics first, only if all occurrences can be arranged to cause, even briefly, control to pass through user supplied code can the user print out the diagnostics he wants rather than the ones someone else wants. This is not a matter of the facilities provided in any particular programming language (an interesting but different topic) but goes to the root of system construction. The picturesque way to put the requirement is that we want to have a subsystem for use by Finns which puts out complete diagnostics exclusively in Finnish without having to put a Finnish option in any central system tables. Any incident which puts out English or anything else counts as undiagnosed. There is no problem except for the class of incident which sometimes cannot be dealt with by the user - e.g. system shutdown or operator intervention. These must, however, be dealt with properly, i.e. in Finnish. This desire for 'complete encapsulation' is quite different from what is done in systems where the handling of faults is dealt with in a unified scheme of levels through which faults are passed back in virtue of a standard unified calling sequence and similar apparatus. In this approach we can have a standard piece of program which is entered by the system after an (untrapped) failure together with data about what happened where. By making assumptions about the general shape of programs it is then possible to construct a sensible diagnostic. A satisfactory job can be done this way, together with a system of levels so that certain faults may only be handled at or above appropriate levels, which gives reasonable information about incidents in programs which obey the standard conventions. It will not, however, provide a proper solution for our hypothetical Finns, because only those incidents could be properly dealt with which one was permitted to handle at the level at which the program ran. We can thus see more clearly the nature of the problem: one general principle says that all diagnostic output should be produced by the user program in formats of the user's choice; the other says that management of a system is only possible if a particular process or job can be inexorably driven out, which is essentially why we forbid some incidents to be trapped or ignored. Whenever a decision is made that certain kinds of fault can never be dealt with internally to a user program but must be passed 'upwards' to some standard part of the system, the consequence is a loss of flexibility which is clear but often overlooked.

Security of information is also involved in this question of dealing with faults internally. If all faults can be dealt with internally, one can rely on the program itself to leave matters in a tidy state no matter what happens. A program whose task is to update or read a sensitive file must ensure that as far as humanly possible it always leaves consistent information, closes the file, leaves no confidential rubbish about in core or registers - even if the user quits in the middle of it, the console plug is pulled out, the multiplexer breaks, the channel breaks, the user's time budget runs out, or the machine-room operator capriciously terminates the user. Once again, if we are prepared to force all programs into a Procrustean bed of convention, external treatments may appear to do the job. But to do this is often both over-restrictive and over-drastic. This security aspect most commonly concerns users updating, by a program for the purpose, data which they are not allowed to read - a very standard situation which is often forgotten or not feasible without special privileges.

Finally, there is the question of robustness. One often makes packages do things for user programs on request - handle disc transfers, for example. These obviously need to handle errors internally, and it is invariably arranged that they can. What is not always dealt with, and in some systems cannot be, is the possibility that the package itself may be catastrophically wrong. Taking a disc transfer package as an example, it will arrange internally to deal with transient checksum failures by trying again. A common class of (software) mistake in programs like this is that they deal correctly with some hardware faults, but rarer hardware error states have never been properly exercised. When the rare event happens the package falls down in a heap, looping or violating its memory bounds or otherwise misbehaving. To the user who called the package it should, nevertheless, appear tidy. That is to say, the user should receive a message

saying 'your requested transfer cannot be done' with as good grace as if it had been a vulgar checksum failure. This can only be achieved if, as well as many other features, there is excellent internal fault handling.

All of these cases are example of how we need not just to know that there is some agency to which we can pass the trouble - but to be able to cope with it on the spot no matter what it is.

The major piece of mechanism should reconcile, or go some way to reconciling, the conflict which occurs between the desire to treat incidents internally to programs and the practical necessity of maintaining supervisor control. It will be seen that the same device can be pressed into service for some of the other needs as well.

A simple mechanism

A simple approach to this is to arrange that all events may be trapped by the user program, but that some of them cause other actions as well. An example of this is furnished by the oddly-named 'private monitor' facility in the Atlas 2 supervisor. This amounts to setting a trap address for all otherwise untrapped incidents, including those which one is not ordinarily permitted to trap. On the occurrence of something which sends control to this address, note is taken by the system of the event. The consequence of this is twofold:

(a) it is not allowed to happen again - instant termination will occur instead
(b) the CPU time remaining for the process before termination is limited to a small amount - in practice 5 seconds.

This gives the moribund operation time to set its affairs in order and ensure that it expires decently. An example is the QUEUEJOB command, whose task is to append the description of a user's job to a file of these things for later running at the discretion of a system operator. The file itself is sensitive - because it may contain users' passwords which get quoted automatically when the jobs are run. It is the task of QUEUEJOB's private monitor sequence to ensure that the file is closed and left in a tidy state and that, for example, no embarrassing information is left available to the ingenious user who quits in the middle of it. The private monitor sequence looks after these things, and system integrity is assured by the knowledge that after a disastrous occurrence the process is certainly on the way out and will terminate before long.

The method described is limited in generality (though it certainly allows one to write systems which say 'STOPPED BY OPERATOR' in Finnish) and one would like to have a method of achieving the same kind of result in more modern systems which do not have the same rigid two-level structure. We need arrangements which will send a process monotonically towards termination or some other standard state, even if it does not get there instantly, while letting it apply its own actions at every stage. This is a strong requirement. The structure of levels through which we proceed monotonically must be known to the system as a whole. Only then can some remedial or tidying action be permitted at each level while nevertheless ensuring that disaster is reported to higher levels with undiminished force. I do not know of any general systems in which all those requirements are properly satisfied; it is easy to arrange for internal tidying at one level only or to arrange that severe incidents are passed all the way back to a high enough level with no opportunity for intervention on the way.

A more general mechanism

The following outline system achieves many of the desired aims, but would probably need hardware help for efficiency.

Each process has at all times at least one 'catastrophe address' which we will call CA. Note that CA is not necessarily concerned at all with the handling of 'ordinary' exception conditions. This is updated (by system call) in either of two ways - by replacing the current value or by stacking a new one on top of it. The stack must be held in space which is ordinarily inaccessible to the process. If a catastrophic incident occurs the following actions take place:

(a) the process's time allowance is set to a period t;
(b) the process's sequence control is reset to CA;
(c) the current CA on the stack is marked as 'used up';
(d) a marker is set which forbids any further use of the calls updating CA;
(e) the process is resumed.

The process continues after re-entry at CA, and continues until one of three things happens:

(a) it issues a system call saying 'end catastrophe program'
(b) t expires
(c) there is another catastrophe.

In many of these cases, the top member of the stack is deleted, and the system behaves as if the catastrophe was new.

The process thus has an opportunity, at any level, to terminate tidily, without having the opportunity to ignore the incident. It does not have full generality - because it is heavily constrained to an arbitrary time t at each stage, and it is not permitted to take precautions against new disasters. But at any rate it can do something, and if the parameters are properly set it can do something useful. The following points are basic to this, and probably to any, solution:

(a) the system must know that a catastrophe has occurred and thus that a process is moribund;
(b) there must be a connection between the actions which occur on exit from a catastrophe program and the reinstatement of the appropriate environment for the new CA. They must be effectively simultaneous.

The more general mechanism just described has not been implemented though the more restricted 'private monitor' system has been in use for some years. It is found an indispensable aid to good diagnostics and to security. Neither approach places any unusual strain on programming language systems, provided that they have some apparatus for dealing with (ordinary) exception conditions. If they do not, they need not be considered further in system contexts. The reason for the lack of strain is that the extra operations required are all part of the enclosing operating system or superior process to which they are just ordinary program. The extra work required at run-time occurs in the mechanism whereby the incident is made known to the affected program, not in the way the affected program reacts. If one wanted to be ideally fancy about it, it would be possible to reserve a name in the language for the kind of incident we are discussing, perhaps 'CATASTROPHE', recognise specially the setting of a condition for it, and compile

extra system calls. This, however, is decorative rather than fundamental. Since the programmer dealing with catastrophes must know what he is about, it is perhaps undesirable as well as unnecessary to bury all the works in the language.

The object of all this mechanism is to arrange that, whatever happens to terminate or impede a process, the various procedures involved can set their own houses in order one by one. Is it in fact worthwhile to go to all these contortions? Surely the answer is yes, particularly in complicated systems involving the integrity of a great deal of data. Present methods are adequate in simple cases, but can easily be overtaken when things get more complex. This is one of the reasons why complex systems with a high integrity requirement tend to be implemented on dedicated equipment. It may be unusual for operators to intervene to kill a process, or for communication equipment suddenly to fail completely, and so on, but it is very important indeed for the right thing to happen if they do. The time has gone by when one could rely on system programmes sorting out messes with tweezers and scalpel - the chaos should never be allowed to arise, and only fault-handling systems of the degree of complexity discussed can cope properly.