Published: 03/14/69

Identification

Standard Error-handling Practice
M.A. Padlipsky, F.J. Corbato´

Introduction

Consider a file directory searching routine which fails to find a specified file: to, say, an editor which has just created a new file, its "not found" result--or "status" --code would mean that there is no necessity to delete an older version; but to, say, the rename command, the same status would mean that the user must be informed of the fact that he´s made an error. (As a matter of fact, the editor should probably treat the "file found" result as a possible error and reflect it to the user.) The point is that in general the decision as to what a particular result from a particular routine "means" should be deferred to as high a level as possible; i.e., to the routine´s caller, or that routine´s caller...or to the user who issued the command which started the whole chain. To put it another way, results must be interpreted and only the caller can interpret them. This is not to say that there are no conditions encounterable which are clearly errors--although the examples which come to mind are usually hardware rather than software conditions: operation not complete, e.g. It is, however, so often the case that the result of a subroutine call is an error only in its caller´s context, that a system-wide point-of-view must be enunciated for dealing with such matters.

Because of the wide variety of conditions which may be encountered, no abstract, absolute rules can be laid down for interpreting status codes. Such decisions must be left to the implementers of the various commands. The handling of error situations after they have been recognized is, however, legislatable. (The result which requires user interaction for clarification must also be left to the ingenuity of the command writer; ipso facto error situations only become evident in such cases when a question must be asked of the user, but the command is currently operating in behalf of an absentee user process.)

There are three broad categories of routines to be dealt with in this discussion: subroutines, commands, and commands which are likely to be called as subroutines. (Some light on the distinction between the first two categories will be cast by BX.0.00, and inferentially by BY.0; the third category should be illuminated by the following.) In addition, a fourth category of routines--subroutine complexes--may be distinguished, so as to cover the special problems of mutually-dependent, mutual-calling groups of subroutines such as the Basic File System "primitives".

Subroutines

The fundamental point about the error handling practice of system subroutines is that results of a subroutine's execution are always referred to its caller for interpretation. That is, subroutines "always" (i.e., barring unforseen, non-software catastrophes) return. Therefore, if there is any potential variation in a subroutine's behavior it always has a "status code" argument in its calling sequence, in order to be able to communicate to the calling routine either the fact of successful completion (i.e., status is "normal") or the encountering of some not-necessarily-anticipated condition (e.g., our old friend "file not found"). The status code argument is declared fixed binary(17), and codes are assigned low, sequential numbers. In general, the codes are unique to the called subroutine; by convention, zero is the code for normal completion. (See the passage below on "subroutine complexes" for discussion of conditions under which status codes are not exactly unique to the called subroutine.) A subroutine's documentation <u>must</u> specify its status code values and definitions.

Note that run-time routines for various languages such as Fortran are frequently a degenerate case of the above, since they do not admit of any completion status other than successful.

Commands

Commands, when they have been invoked as commands, are in general in a position to interpret status codes returned by any subroutines which they call. After testing a non-zero status code returned by a subroutine and determining that execution should <u>not</u> continue (i.e., the code is an error in the context of the command), a command follows the standard practice of invoking the com_err service routine (BD.8.06) as follows:

```
          call com_err (short, long);
```

where *short* is a short-form (character-string) error message and *long* is a fuller version. The com_err routine will check the *brief* option, and place either *short* or *long* in user_output, with the general result that the appropriate message is printed at the user console; if *short* is chosen *long* is stored in <user_error> for subsequent inspection, if the user so desires. The short-form message should be an appropriate 8 character mnemonic, along the lines of those documented in section BY.2.02 which are used by the command system when reflecting file system errors; the mnemonics must, of course, be interpreted in the command's documentation. The long-form message should contain three items: a general description of the error condition (i.e. the interpretation of the short-form mnemonic); specific identifying information where relevant (e.g., if the error condition is "file not found", the file's name); and the offending subroutine's name (if known) and status code. The second item may not be relevant, but if present should be separated from the rest of the message by a semicolon. A contrived example:

```
     call dir_search (name, ptr, code);

     if code=0 go to onward;

     if code >1 go to unknown;

     msg="File not found:"||";dir_search 001";

     call com_err ("nofile", msg);

     onward: return;

     ...
```

Note that the possibility of an unknown status code must be allowed for. (The example assumes that only "1" has been defined.) Standard practice in such cases is to set the short-form message to the null character string and the long-form to the offending routine's name and status code (i.e., for those situations in which the subroutine is reprehensibly returning status codes which are new to the command); com_err will concatenate "Unknown status code:"to the long-form when the short-form is null. When com_err returns, the command returns to its caller.

Commands/Subroutines

It is sometimes the case that routines which are normally commands are often called as subroutines; link and unlink are prominent examples. In such cases, it is clear that the routine must determine which incarnation it is currently in, so as to be able to choose which of the two courses just described to follow. All such commands, and indeed eventually all commands whatsoever, deal with their potentially dual nature in the following fashion: A command named, say, "com" will contain an entry point which has the same name but with an underscore appended (<com>|[com_], or com$com_) for being called as a subroutine. Hence when com is called at <com>|[com], it sets a switch indicating that it has been called as a command; when called at <com>|[com_], it sets the switch to indicate that it has been called as a subroutine. Then, when it becomes necessary to deal with an "error" condition--or, of course,to reflect its own status on completion--it behaves according to the subroutine practice or the command practice as appropriate. Note that this approach imples that the subroutine entry point has an additional argument, for returning a status code. As with subroutines, commands which are intended to be are callable as subroutines must specify their status codes' values and definitions in their documentation.

Subroutine Complexes

The foregoing policies are effective for commands and for what might be thought of as "standard" subroutines. In the Basic File System and the I/O System, however, the subroutines involved are far from standard. That is, the constitutent routines of these two subsystems of necessity make a large number of calls among themselves, and to specially interpret the result of each call according to a set of codes unique to each called routine would be highly inefficient. In a certain sense then, the Basic File System and the I/O System are to be viewed as being each one a single subroutine. The general status information which might need to be reflected by a constituent routine is codified and coordinated, and results encountered "down the line" are in general passed back "up the line" without special interpretation. The responsibility devolves upon the caller of the subsystem to perform whatever interpretation is needful. (Another way of viewing the situation is to raise the quibble that any subroutine in either of the two subsystems can itself return as its status code essentially any status encountered in its subsystem--which is to say that each routine "uniquely" determines its status codes in what happens to be the same way as a number of other routines do.)

In view of the rather complex nature of status codes from subroutine complexes, the command writer must be relieved of as much of the burden of interpretation as is feasible. Naturally, he must test for specific status codes as appropriate to the needs of the command at hand, but each command should not have to account for all status codes. Standard practice in dealing with non-zero status codes from these subsystems must, then, permit pre-interpretation in the sense that status codes are associated with "canned" descriptive information which may then be reflected to the user. In the case of the Basic File System, the information is available through fscodedinfo (BY.2.02). I/O System information is available through check_io_status (BY.4.03). The information gleaned may then be reflected by the command through the com_err mechanism mentioned above.

Faults

There is one class of "error" encounterable when executing a command which is closely related to the issues in regard to commands and subroutines discussed herein. This is the issue of hardware-generated faults. Faults in general are amenable to the standard practice of employing the system condition-handling mechanism, as described in BD.9.04. Fundamentally, this practice is to employ the signal primitive to reflect the occurence of a fault. If the user has established a handler for the condition, that handler will be invoked. Conditions for which no handler is active will be handled by whatever handler is active for the "unclaimed_signal" condition. (The system-supplied default handler for this condition is described in BY.11.05) It should be noted that the condition-handling mechanism offers a particularly flexible and general way of dealing with error conditions, and is available to user programs and subsystems as desired; however, it introduces sufficiently high overhead that it is not employed as standard practice by system commands and subroutines other than in the area of reflecting certain faults (e.g., floating point underflow) to fault-handling routines when appropriate.