To: Distribution

From: Jerry Stern

Date: 11/11/75

Subject: An Extension to the Multics Condition Mechanism

Introduction

This MTB proposes an extension to the Multics condition mechanism that would allow a program to exercise a useful new form of control over the signalling of conditions. This change is appropriately described as a generalization of the existing crawlout mechanism. We shall show that such a feature is extremely useful for solving problems faced by procedures that cannot safely permit an unexpected condition to pass control to a handler defined earlier in the stack.

Due to the relative obscurity of many of the "finer points" of the Multics condition signalling scheme, an overview of signalling is first presented. This is followed by a discussion of the problem that the new feature is intended to solve. Next, the currently available solutions to this problem are examined and shown to be inadequate. After this, the proposed extension is described and its advantages noted. Following this, implementation considerations are discussed.

Overview of Signalling

In this section we discuss the mechanics of signalling as it currently exists. The knowledgeable reader may skip this section. No attempt will be made to discuss the various aspects of signalling at a uniform level of detail. Rather, we shall concentrate on those details that are relevant to the subject of the sections to come, and omit those that are not. The crawlout mechanism, in particular, will be of special interest.

It is assumed that the reader understands the basic facts about condition handling, i.e. how handlers are established and how they are used. The only point worth mentioning here is that a handler is associated with a particular stack frame. When a procedure establishes a handler for a specified condition, an entry is added to a condition list stored in the stack frame for

the procedure activation. This entry contains the condition name and a pointer to the handler procedure.

Conditions are divided into two categories according to their origin. A software condition originates by means of a direct call to the module named "signal_". For example, the signalling of the "command_error" condition is initiated by com_err_ calling signal_. A hardware condition originates by means of a hardware fault, e.g. "zerodivide". When the fault is taken, the "machine conditions" (i.e. the processor state) are safe-stored in ring 0. Control is then transferred to the appropriate fault handling module in ring 0. This module determines if the fault should be mapped into a condition and signalled in the ring in which the fault occurred. If so, a procedure named "signaller" is invoked that manufactures a stack frame on top of the stack for the faulting ring. The machine conditions are copied into this frame. A transfer is then made to the signal_ procedure that executes in the faulting ring and appears to have been called from the manufactured stack frame.

As just described, both software and hardware conditions eventually result in a call to signal_. It is this module that actually implements the signalling mechanism. Beginning at the top of the stack, signal_ examines the condition list of each stack frame. It looks first for the particular condition name specified as one of its arguments. If no handler is defined for that condition, it then looks for the "any_other" condition. Either way, if a handler is found, that handler is invoked by signal_. Otherwise, signal_ skips to the next frame down the stack and continues the search.

All handlers called by signal_ have a standard argument list. Among the arguments is a pointer to the machine conditions structure. This pointer will normally be null for software conditions. Also included in the arguments is the "continue" flag. This flag is initially set off. If it remains off when a handler returns to signal_, then signal_ will return to its caller. If its caller is signaller, an attempt will be made to restart execution from the point of the original hardware fault. This is all done by black magic and will not be pursued here. If the continue flag is turned on by a handler, then when the handler returns, signal_ will continue its search towards the bottom of the stack invoking other handlers as they are found.

If the bottom of the stack is reached (either because no handler was found or because all handlers found set the continue flag) one of two actions is performed. If this is really the absolute bottom of the logical process stack, i.e. no outer ring stack segment exists on which we can continue signalling, then the situation is hopeless and the process is terminated. If, on the other hand, the bottom stack frame is threaded to another frame on an outer ring stack, then a crawlout is performed.

A crawlout is designed to exit a ring as gracefully as possible and to arrange for signalling to continue on the outer ring stack. Once an exit is made to the outer ring, it is not possible to return to the inner ring due to the nature of the ring protection mechanism. Therefore, a crawlout is inherently one-way, i.e. restarting from the point of origin of the condition becomes impossible.

The crawlout mechanism is conceptually simple, although the details are a bit messy. The first step of a crawlout involves "unwinding the stack". Basically, this amounts to searching the stack from top to bottom a second time looking for a special type of handler for the "cleanup" condition. Each cleanup handler found is invoked. (1) The next step of a crawlout is to manufacture a stack frame on top of the outer ring stack on which we want signalling to continue. This frame is known as the "signal caller frame". The machine conditions and other arguments to signal_ are copied into this frame. Finally, the last step of the crawlout is a special (ALM-assisted) transfer to a procedure executing in the outer ring that appears to have been called from the signal caller frame. This procedure is none other than signal_ itself which proceeds to signal the original condition on the new stack. The signal caller frame is cleverly set up so that if signal_ ever returns (as it might if a handler returns with the continue flag off), the call to signal_ is repeated. This is necessary since, as mentioned above, a return to the inner ring is not possible.

The Problem

Let us define a "critical operation" to be an operation that should not be interrupted by an unexpected condition. An operation may be termed critical for a variety of reasons. For example, the operation may require locking a resource that should not stay locked indefinitely. Alternatively, the operation may require unusual modifications to the normal process environment, e.g. masking ips signals, changing standard I/O switches, etc. The danger here is that control may pass to some handler defined earlier in the stack that cannot (and should not) be assumed to understand the critical nature of the interrupted operation. Consequently, this handler may benignly commit some grievous error due to its necessary ignorance of the situation.

It becomes apparent that, in view of the dangers, conditions occurring during a critical operation must be intercepted by the procedure performing (or initiating) the operation. We shall

---

(1) This same mechanism for unwinding the stack is also used whenever a non-local goto is executed.

call this procedure the "critical procedure". By hypothesis, the condition is unexpected and hence cannot be "handled" in the conventional sense, i.e. there is nothing the critical procedure or any handler established by it can do that would safely allow the critical operation to be restarted. Therefore, the only recourse is to restore the environment to the state that existed before the critical operation began. Or if this is not entirely possible, then at least the environment must be restored to some "secure" state. Having done this, it would then presumably be safe to allow signalling to continue. However, there is still a danger that some handler might return, thereby causing the critical operation to be resumed from the point of origin of the condition. This is clearly unacceptable once the special environment has been restored to normal, i.e. locks unlocked, masks unmasked, etc. If the critical operation is to be restarted, then it must be restarted from the beginning and not from the middle.

Before proceeding to discuss the available remedies to this problem, let us first show that this is, indeed, a real problem by giving a real example. The message segment facility provides one such example. The central procedure of this facility, called "mseg_", is a critical procedure responsible for the locking and unlocking of message segments. Clearly, a message segment should not be permitted to remain locked indefinitely since various system services depend on shared message segments. Some message segment operations are performed entirely by mseg_, while other operations are performed by subroutines called by mseg_. In all cases, however, mseg_ must ensure that the message segment is locked before the operation begins and unlocked when the operation completes. If an operation is interrupted by a condition, then mseg_ must ensure that the message segment is salvaged and unlocked before control is relinquished.

It is in this last duty that mseg_ fails miserably. The approach taken by mseg_ is to establish a cleanup handler to do the salvaging and unlocking. In effect, mseg_ is counting on the fact that it is an inner ring procedure. It is also counting on the fact that none of the procedures in the chain of calls from the message_segment_ (or mailbox_) gate to mseg_ establish any handlers. Given these circumstances, a crawlout is inevitable whenever a condition is signalled, and hence the cleanup handler established by mseg_ will, in fact, be invoked before any harm can be done.

Unfortunately, mseg_ is not always an inner ring procedure. Certain system processes, e.g. Backup, run in the same ring as mseg_ and hence no inward call is ever made. In these processes, if a condition occurs during a message segment operation, no crawlout will occur. Instead, it is likely that the default handler established by the process overseer will be invoked. The normal action taken by the default handler is to come to command

level, thereby leaving the message segment locked indefinitely!

The ring problem afflicting mseg_ is by no means an essential aspect of the critical operation problem. Any ordinary user procedure can perform a critical operation involving a locked data base. However, the case of mseg_ is especially interesting because of the approach taken to solve the problem. This approach wrongly depends upon the occurrence of a crawlout. When, in fact, a crawlout does occur, however, the approach works quite nicely. As we shall see later, this idea has considerable merit.

Available Solutions

Clearly, in order to prevent conditions from "escaping", a critical procedure must establish an any_other handler to intercept all unexpected conditions. As discussed earlier, this handler can do nothing that would permit the critical operation to be safely resumed. It must therefore restore the environment to a "secure" state. The question is what to do next. We would like to continue signalling because, if nothing else, this is the best way of communicating precisely what has gone wrong to some higher authority. As already pointed out, however, this is unacceptable due to the fact that some other handler may decide to return to signal_ with the continue flag off. This would cause the critical operation to be resumed after the necessary environment has been undone. We must also consider the possibility that the condition may not have occurred within the critical procedure itself. It may, instead, have occurred within some subroutine called by the critical procedure. The critical procedure is not necessarily in a position to restore the subroutine environment. Therefore, it is preferable that the subroutine be given a chance to do this using the normal cleanup mechanism.

One possible approach is to attempt to return to the caller of the critical procedure in a normal manner. This could be accomplished by executing a non-local goto from the any_other handler to an appropriate location in the critical procedure. Happily, this would have the desired effect of invoking all cleanup handlers defined in the stack above the critical procedure. However, this approach is only reasonable if there is some way to indicate to the caller of the critical procedure that the operation was aborted, e.g. by setting an appropriate error code. Already we can see that this approach is non-general. There may be no error code or other method of indicating an error. This is because error codes and the like are designed for anticipated errors, of which there may be none. (The condition mechanism, on the other hand, is designed for unanticipated errors; but we can't seem to use it here.) Of course, we could insist that all critical procedures have error code arguments for

this purpose alone if none other.

Let us assume that an error code argument exists. Does this then present a reasonable solution? Unfortunately, it does not. If for all possible conditions, we simply set the error code to some universal value, e.g. "unexpected_condition", we have lost all traces of what went wrong. We could try to be fancier, i.e. map each different condition into a different error code, but this imposes an unreasonable burden. Perhaps some system-provided utility procedure could do the mapping. Still, this would not account for user-defined conditions. More importantly, no matter what error code is returned, we have lost the machine conditions (and other information transmitted by signal_) which are vital to diagnosing the trouble. This is not merely a matter of providing debugging information. Sometimes the user, playing the ultimate handler, must himself take action based on information communicated by signal_ (and printed by default_error_handler_). For example, if a record_quota_overflow condition is signalled, the user must know which segment was being referenced so as to determine the directory that is out of quota.

It would seem that what we need is a way to allow signalling to continue without the danger of restarting the critical operation. There is, in fact, a rather underhanded method that will accomplish this feat. First, the any_other handler must copy the machine conditions and other arguments to signal_ (available from find_condition_info_) into static storage or space reserved in the stack frame of the critical procedure. Then the handler can execute a non-local goto to a special location in the critical procedure. At this location the necessary environment restoration can be done and the any_other handler reverted followed by a call to signal_. This call will use the stored arguments to the original call to signal_, thus faithfully resignalling the condition. The call to signal_ can be embedded in a loop so that if it ever returns, it will simply be repeated.

As already admitted, this solution is a hack. It uses the signalling mechanism in a manner that was not intended. As a result, the condition will appear to have originated at the special location in the critical procedure where the call to signal_ is made. But the machine conditions and other signal_ data will, in fact, apply to the true origin of the condition. This minor inconsistency is not so terrible. What really makes this a hack is that it requires too much knowledge of the signalling mechanism. The technique would probably not even occur to the average programmer. Indeed, it did not occur to the author until carefully studying the signalling mechanism. Nor did it occur to several other experienced Multicians to whom the author posed the problem. But even if the technique were widely known, it would still be undesirable because it forces the programmer to concern himself with details that are irrelevant to the problem. Nevertheless, despite this drawback, the technique

does produce the desired effect.

Proposed Solution

The clever reader will already have noticed that the technique just described is essentially equivalent to a crawlout. In both cases, the arguments to signal_ are copied into a new stack frame, the stack is unwound, and a transfer is made that results in signal_ being called anew. Therefore, it is proposed that the crawlout mechanism be generalized in such a way that it can be utilized to solve the problem at hand.

The key to generalizing the crawlout mechanism is to define an appropriate abstraction that the programmer can understand without irrelevant details. If we view the collection of stack segments from different rings as a single logical stack, we note that there are implicit boundaries that exist between adjacent stack frames in different rings. In the course of signalling a condition, if one of these boundaries is encountered, a crawlout must be performed across the boundary. A simple extension to this scheme is to allow new boundaries to be defined explicitly. We shall call these boundaries "condition walls".

Given this new tool, it is relatively easy for a critical procedure to protect against unexpected conditions. Essentially, all that is necessary is to first establish a condition wall and then to establish a cleanup handler to do the necessary tidying up. Any condition that reaches the condition wall will cause a crawlout to occur, thereby invoking all cleanup handlers defined above the condition wall in the stack. Note that the condition wall is created immediately below the stack frame of the critical procedure. This allows all outstanding procedure activations to be cleaned up uniformly using the standard cleanup mechanism. It is not necessary for the critical procedure itself to be cleaned up by an any_other handler (which always seemed rather inelegant).

Implementation

Implementation of the condition wall feature is fairly simple. A new procedure, "establish_condition_wall_", must be provided. This procedure will simply turn on a bit in the stack frame of its caller indicating the existence of a condition wall. The signal_ program must be changed to check this bit at the same point it now checks for implicit condition walls, i.e. the jump to a stack frame on an outer ring stack. In either case, the operation of the crawlout is basically the same with one notable exception.

When a regular inter-ring crawlout is performed, signal "pushes"

the signal caller frame onto the top of the outer ring stack. When an intra-ring crawlout is performed, however, the new top of stack, i.e. the condition wall, is in the middle of the stack. On top of the stack is the frame for the activation of signal_ that is executing the crawlout. Needless to say, this is a delicate operation and some care must be exercised to make it work. We cannot simply build the signal caller frame in place, i.e. immediately above the condition wall, unless we know there is enough room. Unfortunately, there is no room. Any of the frames above the condition wall may contain data that is part of the arguments to signal_. In other words, we might inadvertently overwrite the very data we're trying to copy into the signal caller frame. Therefore, the recommended approach is to have signal_ extend its own stack frame and build the signal caller frame in this space. Then, at the last moment, the signal caller frame can "slide" down the stack into position.

This technique need not be special to intra-ring crawlouts only. It can be used for all crawlouts. In fact, we can even use this technique to fix what might be considered a bug in the current crawlout implementation. This bug derives from the fact that signal_ unwinds the stack before building the signal caller frame. In unwinding the stack, a cleanup handler might be invoked that could inadvertently destroy the arguments to signal_ before we have copied them. For example, imagine a program that allocates a structure and establishes a cleanup handler to free it. A pointer to this structure may be passed as an argument to signal_. Thus, in the event of a crawlout, the structure will be freed before signal_ copies it. One might consider this to be a programming error, although understanding why and how this is an error is probably beyond the grasp of the average programmer. In any case, the problem is eliminated if signal_ first builds the signal caller frame in its own stack frame, then unwinds the stack, and then moves the signal caller frame into place.

The signal caller frame has a special crawlout flag turned on. This flag is copied by find_condition_info_ into the "cond_info" structure that it returns. In order to distinguish the two different types of crawlouts, a second flag, call it "explicit_crawlout", should also be kept both in the stack frame and the cond_info structure. If nowhere else, this flag must be checked by default_error_handler_ in order to print an appropriate crawlout message.

Conclusion

The proposed condition wall feature solves in an elegant and appropriate manner a general problem for which no adequate solution currently exists. Good programming practice is encouraged by this feature in contrast to the kludgery now necessitated by its absence. The proposed feature is economical

in the sense that it makes use of an existing system mechanism with only a few minor changes.