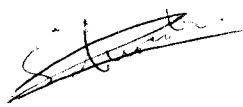


file -
Multics
papers

TO: R. Freiburghouse
J. Gintell
N. Morris
✓ J. Saltzer
S. Webber



FROM: R. Feiertag, V. Voydock

DATE: April 5, 1972

SUBJECT: Non-Local Transfers, Ring Aborts, and Outward Ring Transfers
on the Multics Follow-On System

This document describes the design changes necessary for the implementation of the above-named features on the follow-on system.

A Ring Abort (Crawl Out) Mechanism for the Multics Follow-On System

The current Multics system contains only one means of abnormally aborting a ring. Such an abortion occurs only if the condition mechanism cannot find an established handler for a signalled condition in the ring of execution. This form of ring abort (currently called crawling out) must be maintained in the Multics follow-on system. The design proposed below also contains provisions for a new form of ring abort, the non-local transfer. A non-local transfer to a label in a higher ring will also cause ring abort and will actually be the heart of the implementation of the follow-on crawl out mechanism.

The Non-Local Transfer

Since the non-local transfer, as implemented in the unwinder, forms the heart of the implementation of the ring abort mechanism, it will be described first. The current unwinder makes use of some idiosyncracies of

the return sequence. The change in the return sequence makes it no longer possible to implement the unwinder entirely in PL/I. Also the current unwinder can cause some degenerate race conditions (e.g., quitting after the unwinder reverts the cleanup procedure and before it invokes it) which will result in undesirable effects. The probability of the occurrence of these race conditions can be sharply reduced in ALM. For these reasons the new unwinder should be written entirely in ALM. However, the program will be fairly small.

The unwinder is invoked with a label as an argument. The unwinder will first perform some consistency checks on this label. The stack frame pointer in the label must point to a 16 word aligned address (all stack frames must be 16 word aligned), the stack frame pointer must point to a valid stack segment (this can be validated by looking for a proper stack header), and the forward and backward pointer in the indicated stack frame must also be 16 word aligned. These simple checks will uncover almost all cases of invalid labels before the unwinder starts unwinding the stack and destroying valuable information and very possibly terminating the process.

The unwinder now begins examining each stack frame in the stack beginning with the most recent. For purposes of discussion, it is assumed that unwinder does not initially create a frame of its own, however, when implemented it may be more convenient to do so. The unwinder must keep the procedure pointer and stack frame pointer in the label given it in pointer registers.

For each stack frame, the unwinder first checks to see if this is the target stack frame. If so, the unwinding is complete and the unwinder simply transfers to the location indicated in the label. Before performing this transfer, the unwinder must restore enough of the language environment to allow the target procedure to run. What is restored depends on the language; the unwinder must have builtin knowledge of the run-time environment of every language that invokes it. For PL/I version 1 the ap register must be restored. If this frame is not the target frame a check is made for a cleanup procedure established in this frame. If one is established it must first be reverted and then invoked (note that invoking the cleanup procedure will require doing a push and saving the label stored in the pointer registers). When the cleanup procedure returns, the unwinder pops its stack frame so that it is again operating in the frame of the activation being unwound. Now the unwinder places a pointer to itself in the return pointer in the stack frame preceding the one being unwound and a normal return is performed. This serves two purposes. It destroys the stack frame just examined and causes the unwinder to continue operation in a different ring if the next most recent stack frame is in a different ring. The latter is a consequence of the operation of the rtd instruction. In this way unwinding across a ring is performed automatically without special code. Now the unwinder begins examining the new frame.

*Point to the
unwinder
how it will
be done*

The above algorithm is only one means of accomplishing the unwinding process, but does illustrate the necessary functions which must be performed. Particular implementations may vary in exact technique.

Automatic Ring Abort

When the condition mechanism cannot find a handler for a signalled condition in a given ring an automatic ring abort is performed and the condition is again signalled in the calling ring. This feature will continue to operate on the follow-on.

The design outlined below assumes three important decisions. First, the machine conditions of the wall-crossing fault at the time of the inward call which resulted in the ring abort will no longer be returned to the calling ring. This is necessary because the wall-crossing fault no longer occurs on the follow-on machine. Secondly, a handler of a condition which has already resulted in a ring abort cannot return to the condition mechanism and expect it to reexecute the inward call as is currently the case. This is also the result of the lack of an inward wall-crossing fault, because the condition mechanism no longer knows what the inward call was. Finally, the condition mechanism will, after a ring abort, continue to pass to the handler, the condition name, and machine conditions, if any, associated with the original raising of the condition in the inner ring. It is not clear that this is not an access violation,

because, an outer ring does not necessarily have the right to know about anything that occurred in an inner ring. The machine conditions may contain sensitive information as contained in the machine registers and executing instructions. This policy of passing the condition name and machine conditions of a condition raised in an inner ring is being continued for ease of debugging. However, this policy should be reviewed at a later date.

When a condition is raised, the condition mechanism searches for a handler by examining each stack frame, starting with the most recent frame, looking for a handler established for that condition. As it examines each frame it checks the stack frame back pointer to see if that pointer contains a ring number greater than the current ring. If the ring number in the back pointer is greater than the current ring then this stack frame is the earliest in this ring and if no handler for the raised condition has been found then an automatic ring abort is initiated. The condition mechanism constructs a dummy stack frame on the calling ring stack (the calling ring stack is determined from the back pointer containing the ring number greater than the current ring). Into this dummy frame is copied the condition name and machine conditions associated with the raised condition. A label is constructed containing a pointer to the dummy frame as the stack frame pointer and a pointer to a special ALM procedure as the procedure pointer. The unwinder is then invoked with this label. As explained

earlier the unwinder will perform the ring abort and will finally call the special ALM procedure in the calling ring as indicated in the label. This ALM procedure will fabricate an argument list for a call to `signal_` giving the condition name and the machine conditions copied into the dummy frame. This procedure will then call `signal_`. Thus the automatic ring abort is complete. If `signal_` returns to the special ALM procedure it will signal a special condition indicating an illegal attempt to restart an operation which resulted in a ring abort. The direct call to `signal_` after the ring abort causes problems. For certain conditions the `pjl signal` statement generates a call to a procedure which takes special action before calling `signal_`. The signalling mechanism has no knowledge of this procedure and thus will not invoke it after a ring abort. This is a special case of a general problem; we do not plan to solve it at this time. Note that the problem already exists in the current crawlout mechanism. It is not currently a problem since the rings in use (1 and 0) do not use the conditions which cause this problem to occur.

Outward Ring Transfers

The current Multics system supports a simple outward ring transfer mechanism which allows a procedure to invoke another procedure in a higher ring provided no arguments are passed. The target procedure is not permitted to return and all active rings lower than the target ring are aborted. This form of ring abort is not a normal ring abort in that no cleanup

procedures are invoked. This limited form of ring abort is only a temporary solution and should be expanded to perform a full ring abort for all currently active rings. However, the means of doing this are not obvious at this time, but should be explored at a later time. The Multics follow-on system will support only the limited form of ring abort for outward ring transfers initially.

On the follow-on machine, an attempted outward call will result in a fault that will be handled by the gatekeeper. The gatekeeper determines the target ring by looking at the ring brackets of the target procedure. It then sets the stack end pointer of each existing stack associated with a ring between the faulting ring and the target ring inclusive to be equal to the stack begin pointer in cases where that is not already the case. This logically truncates the stacks in those rings. The gatekeeper then zeroes the registers and nulls the pointer registers in the machine conditions associated with the outward call fault, since they contain privileged information. The gatekeeper then sets pointer registers 6 and 7 to point to the base of the target ring stack, (the format of the stack header is such that this will cause the target procedure to establish its frame at the beginning of the stack). It sets pointer register 0 to point to a zero length argument list. It then sets the procedure ring register of the machine conditions to the target ring and returns to the fault interceptor which restores the altered machine conditions and completes the outward call.