M.I.T. Laboratory for Computer Science
Computer Systems Research Division

May 18, 1976
Request for Comments No. 113

This problem
looked in some length
by a
forward thinking
program semantics
type

# Signalling for Subsystems

by Harold J. Goldberg

This RFC describes two problems found in the current Multics signalling mechanism, with respect to consistent handling of inter-subsystem signalling. A solution is proposed that eliminates the need for ad hoc solutions to one problem, and provide a needed solution for the other. I would particulary like comments on the model I chose for my solution. I do not have a model for the solution to the first problem I describe, but it fits very well with the implementation for the second problem.

The first problem deals with entering and exiting subsystems via the signal mechanism. Consider a subsystem <S> that provides some abstraction to its user <U>. <S> attempts to provide a clean interface to <U> including the shielding of <U> from errors that <S> can handle internally. There are times, however, that <S> must let <U> know about an error that <S> cannot handle. There are two categories of this type of error. They are 1) recoverable, and 2) non-recoverable. The error is broken down according to whether or not <S> wants to continue after the error, assuming that something has been externally fixed. Jerry Stern's "cleanup wall" provides a solution to the

This note is an informal working paper of the M.I.T. Laboratory for Computer Science, Computer Systems Research Division. It should not be reproduced without the author's permission, and it should not be referenced in other publications.

non-recoverable kind of error [MTB 236]. Of interest in this paper is the proper handling of the recoverable errors.

To demonstrate the problem, consider that <U> has called <S> which called an internal subroutine <I> that discovered an error condition. <I> will then signal this condition. This state of affairs is shown in figure 1. The program <SIGNAL> implements the signalling mechanism in this model. By the usual means, <S>'s condition handler is invoked, and is labeled <S'> in figure 2. <S'> does not know how to handle the condition and therefore wishes to continue the signalling of the condition. However, before this is done, <S'> wants to fix up the global state of the process so that programs outside of the subsystem <S> may run correctly. An example of this might be that <S> has put the terminal output stream (user_i/o) in raw output mode so that it can produce graphic images. Should a program outside the <S> subsystem attempt to print an error message in this mode, it would not be understood by the reader. Thus, <S'> fixes up the output stream and then does what is necessary to continue the signalling. In this way, subsystem <S> has been exited via the signal mechanism.

Now the next condition handler is invoked, and is shown in figure 3 as <U'>. <U'> fixes the exception and then returns. By returning to <SIGNAL>, a "restart" operation is performed. Thus, <I> continues its execution and hopefully will succeed now. However, by restarting in this way, the subsystem <S> has been re-entered without <S>'s knowledge. Thus, the terminal output stream would not be reset to raw output mode again, and now anything "graphic" that <S> will attempt will fail.

Maybe signalling is the wrong semantics? You really should back off <S, and re-enter from the start! (even return)

One ad hoc solution used to solve this problem is pictured in figures 4 and 5. <S'> wants to remain on the stack so that it can regain control before <I> is restarted. However, <S'> also wants to continue the signalling of the condition. <S'> does this by first setting some internal static indicator "ignore" and proceeds to signal the condition over again. <S>'s handler gets re-invoked as <S''> and checks the ignore indicator. Since it is on, <S''> just does what is necessary to continue the signal and returns. Finally <U>'s handler, <U'>, gets invoked (see figure 5). When <U'> returns, <S'> will get control. <S'> will then reestablish the appropriate environment for <S>, turn off the ignore indicator, and restart.

Since this set of operations might be necessary in many cases, it would be wise to define a specific method of performing them. A subroutine might be called to do this, for example. The solution that was chosen solves both problems presented in this paper, and thus will be discussed after explaining the second problem.

The second problem in the current Multics signalling mechanism deals with the chain of stack frames that are searched when looking for a condition handler. By searching each frame succesively from the top of the stack, a handler for a condition related to a specific subsystem might be invoked without knowing anything about the circumstance at hand. To make this point clear, consider the situation depicted in figure 5 again. Control has finally passed to <U'> to handle a condition internal to the subsystem <S>. How this might happen has already been explained. However, if <U'>, or any program that <U'> calls, detects an error, by the current mechanism <S>'s handler will be invoked once again. One could argue that <S>'s handler has turned its ignore indicator on and would therefore pass the signal on, but then consider what would happen if <I> has a handler for the new condition that <U'> signalled. It is not <I>'s business to handle <U'>'s errors, but it would none the less be invoked by the current mechanism. What would be desirable in this case is for <I>'s handler to pass on the signal too.

Once a handler for a condition decides to pass the condition on, all the stack frames that lie between the handler's stack frame and the frame where the handler was establised should become dormant with respect to conditions that arise from the passing of the condition. This dormant state should remain in effect until that handler is finally returned to. What this means in terms of figure 2 is that after <S'> decides to continue the signal, those frames between <S'> and <S> should be avoided when signalling future conditions, until <S'> is returned to.

## Implementation

The implementation that the author has conceived is derived from the conceptual model that all subsystems are one large program containing all necessary programs as internal subroutines. Condition handlers are likewise considered internal subroutines (they are in fact implemented as such). If an error occurs in an internal subroutine, the proper stack frames to look for condition handlers in are the parents of the internal procedure. One should not expect that an external subroutine, which does not know the internal workings of the module that took the fault, is able to handle an error condition caused by a module, better than the module itself. The situation is shown in figure 6, where <S> is the subsystem with <I> as an internal procedure. <X> is some arbitrary procedure, which might in fact be part of some other subsystem. Once again, the chain that should be followed if a condition is signalled by <I> or anything that <I> calls is as follows: First <I> will be checked for a condition handler, and then <S> will be, whereas <X> will be bypassed. If one claims that <X> was the intended recipient of the signal, then <I> should not have been an internal procedure to the module <S>.

To implement a mechanism that solves the two problems described, the notion of "logical parent pointer" is used. The logical parent pointer is an extension to the physical parent pointer, or display pointer. It is assumed that there exists such a pointer in every stack frame. In signalling, first the real PL/I display pointer of a frame is used to find the next frame to look at. This step insures that a physical parent is always the logical parent. If there is no real display pointer, the frame's logical parent pointer is examined. A zero or null logical parent pointer means that this frame is not logically part of any subsystem, and thus the proper place to signal is on the callers frame (the immediately preceeding frame). Since subsystems are not always compiled as one program, we asume that separately compiled programs that are logically part of a subsystem must set their logical parent pointer explicitly. (1)

(1) Neither the entry sequence of programs, nor the building of stack frames need be changed, since a bit that is always made zero upon frame setup can be used to indicate the validity of the logical parent pointer. This approach is currently used to indicate the validity of a frame's condition list.

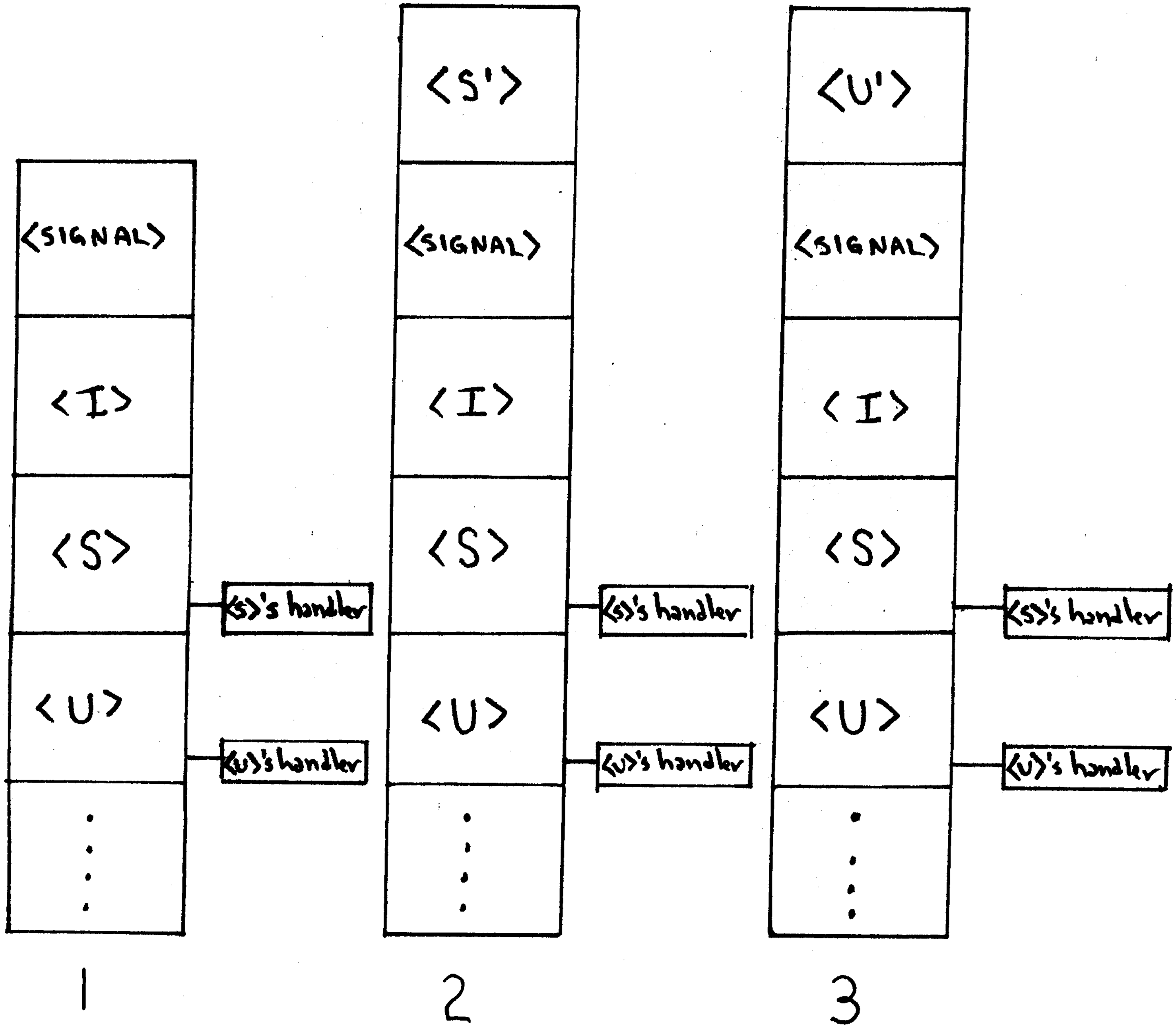
So far this does not explain how the problems are solved. To solve the problem of re-entering a subsystem without the knowledge of the subsystem, we invent the procedure "<CONTINUE>". <CONTINUE> is called by any handler who has been invoked and wants to continue signalling. As an example consider figure 7. By having <S'> call <CONTINUE>, <S'> is insured that its activation record remains on the stack and is re-entered when and if the condition is satisfied. The important step comes now. <CONTINUE> sets its logical parent pointer to point to the logical parent of its callers's logical parent (i.e. the logical parent of <S'>'s parent). It then signals the condition over again. By setting its logical parent pointer in this way, any further signals will not be noticed by all of those frames between <CONTINUE>'s frame and <S'>'s parent inclusively. If <S'>'s parent was the outermost procedure of a subsystem, the entire subsystem activation is removed from the signal chain until <S'> is returned to.

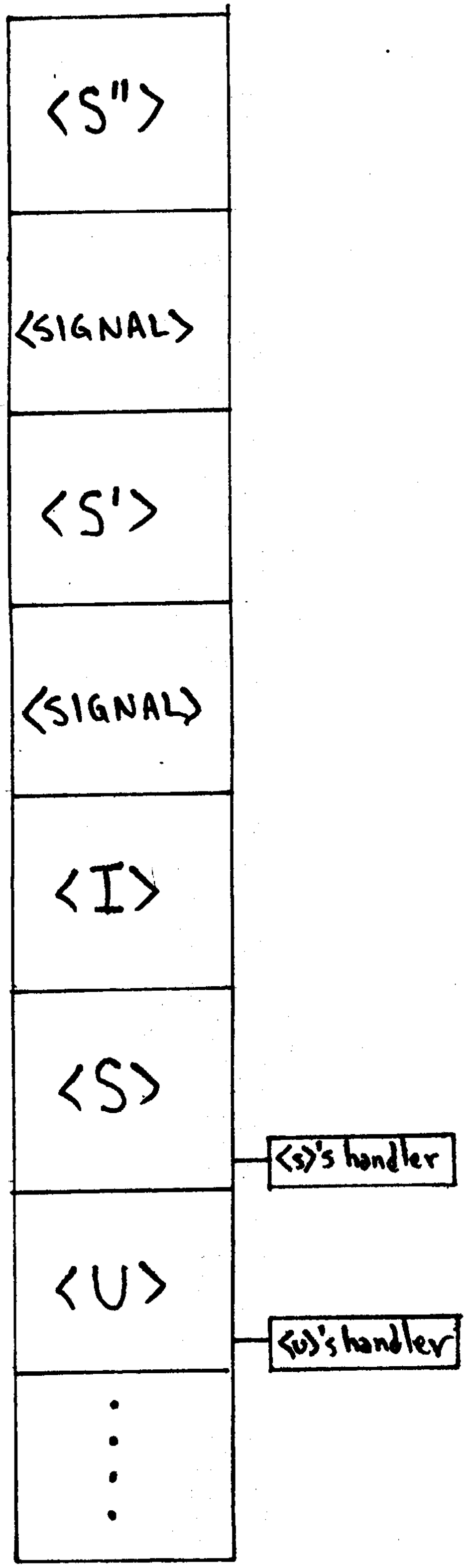
Any handler that gets invoked after this set of events will have a real display pointer pointing to something equal to or even further down the stack than <CONTINUE>'s logical parent pointer. In this way, increasingly larger sections of the stack are sectioned off as they become dormant during error handling.

A final observation is that the special signal "cleanup" could not use this signalling mechanism, and is the only mechanism that should look for handlers sequentially from the top of the stack downward. This is because the cleanup signal was intended to inform all intevening stack frame owners that they will no longer be returned to.

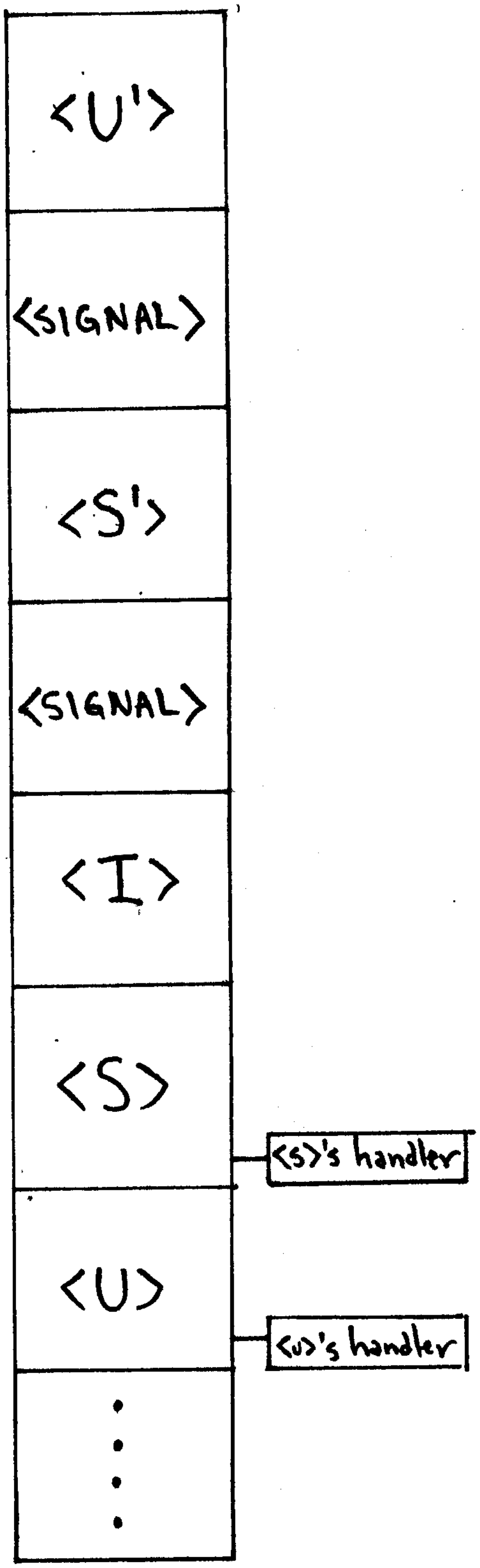
## Acknowledgement

I would like to thank Dave Reed for help in identifying the specific problems and with developing the proposed solution.

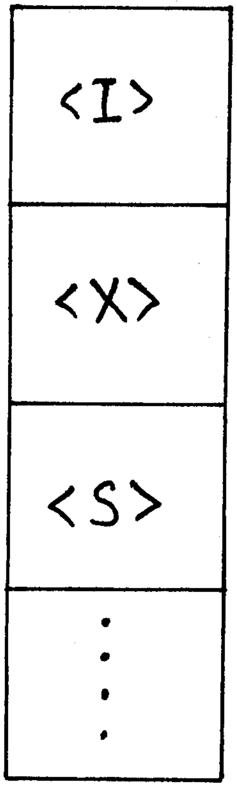

<SIGNAL>
<I>
<S>
<S>'s handler
<U>
<U>'s handler
1
<S'>
<SIGNAL>
<I>
<S>
<S>'s handler
<U>
<U>'s handler
2
<U'>
<SIGNAL>
<I>
<S>
<S>'s handler
<U>
<U>'s handler
3



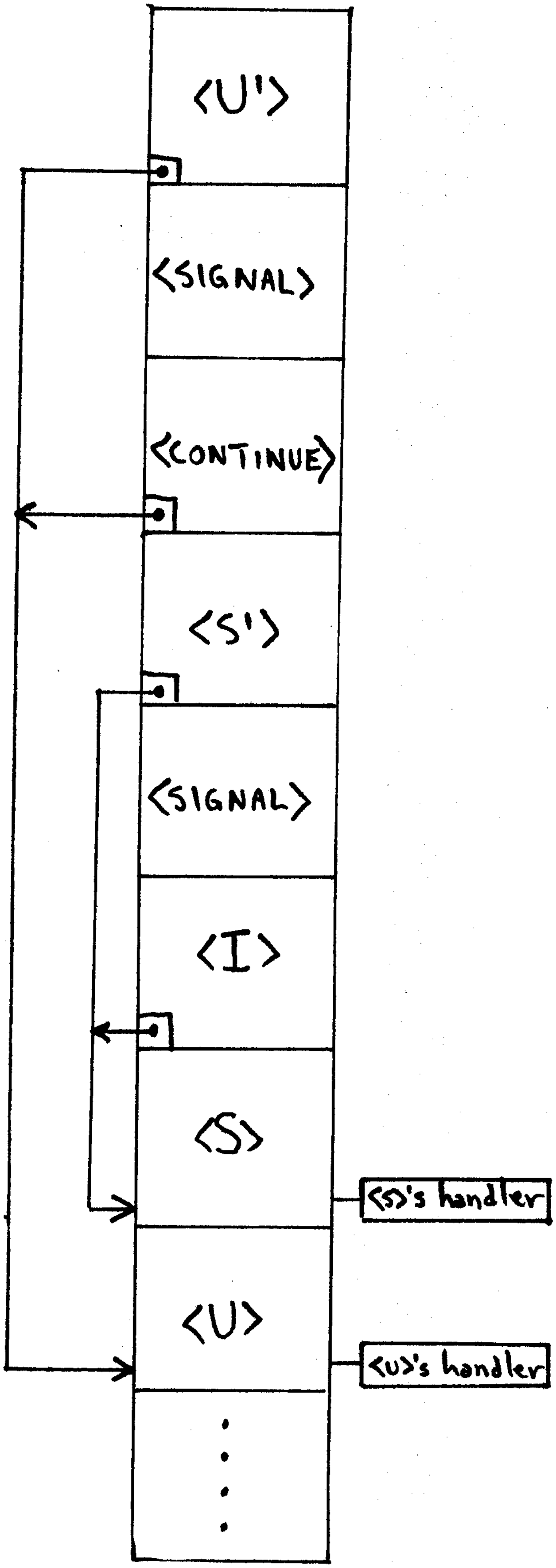
4



5



6



7