Published: 06/30/67

Identification

Abnormal Returns: The Unwinder
R. M. Graham, M. A. Padlipsky

Purpose

In some fairly basic sense, a Multics process is a sequence of subroutine calls and returns. Indeed, system-standard call, save, and return sequences are crucial to the functioning of the System (see BD.7.02, BD.7.03). Occasionally, however, it is necessary to exit from a subroutine by transferring to a label which was furnished as an argument, or which is available in some known fashion. Such exits, which do not employ the standard return sequence, are known as "abnormal returns" (in EPL, "non-local go to's" are implemented as abnormal returns). The present section describes the Multics mechanism for effecting abnormal returns, "the Unwinder".

Overview

The following discussion assumes the reader is familiar with Multics condition handling as described in BD.9.04. There are two basic problems which arise when a abnormal return is attempted. First, there is the issue of protection: some provision must be made to prevent transfering off to a label regardless of what ring the procedure containing the label resides in. Second, there is the issue of unfinished business: for the procedure being left and for the procedures being bypassed, call-save-return Stack frames must be released, EPL/PL/I "epilogues" must be executed, and, indeed, whatever tidying up the procedures involved have to do in general must be provided for as well. The Unwinder, described herein, undertakes to solve these problems. Let us, then, consider the design of the "unwinding" scheme in the abstract, for the solutions to the basic problems are implicit in it.

Abnormal returns are handled in Multics in a fashion analogous to that in which conditions and signals are. To prepare for abnormal returns, a procedure invokes the condition primitive (BD.9.04) in order to place a "pseudo-handler" on a special push-down list (called cleanup) which will be employed when the abnormal return is effected; the pseudo-handler is a procedure which takes care of one or more items of the unfinished business mentioned above.

To effect an abnormal return, a procedure calls the system procedure unwinder with the label to be abnormally returned to as argument; the Unwinder, a ring-0 routine, will invoke procedures which have been placed on cleanup lists before effecting the abnormal return. Both the preparation and the effecting deserve and will receive more detailed discussion; but in very broad terms, the preceding two sentences are "all there is" to abnormal returns: put a procedure-to-be-executed-in-the-event-of-abnormal-return in a fixed place, and make the abnormal return by means of the Unwinder.

The cleanup "stack" is kept in the signal vector, along with all the condition-handler lists which are established by condition. (Note that in actuality there is a signal vector for each ring, and "the" signal vector is a "logical" entity.) Cleanup may be viewed, as a matter of fact, as just another condition. However, there are certain important differences between cleanup and, say, overflow: Overflow will be "signalled" by an invocation of the signal primitive (BD.9.04 again), cleanup will be "signalled" by an invocation of unwinder; indeed, signal specifically rejects "cleanup" as a condition name. More important, signal will invoke in turn only the most recently established handler for the overflow condition; unwinder, on the other hand, may invoke several established handlers for the cleanup "condition"; at least, it may invoke them provided that they all terminate in normal returns to it. Obviously, placing handlers on the cleanup list can lead to complicated mistakes; let the user beware. The heaviest use of the cleanup "condition" is expected to come from compilers which allow a PL/I-like "block" structuring. At any rate, to specify that procedure proc is to be executed in the event of an abnormal return from the current procedure (or an abnormal return past it, from a routine which it has called)

```
     call condition ("cleanup", proc);
```

The Unwinder itself plays a role analogous to that of signal/signal_search. It is invoked as follows:

```
     call unwinder (lbl);
```

where lbl is a label (most probably passed to the invoking procedure itself as an argument), which if in another ring, has been established as a "door", in the sense of BD.9.00 (see also below). The first task of the Unwinder is to invoke any procedures which are on the cleanup stack in the signal vector of the ring it was invoked from (say <signals_n>) and possess the current invocation number.

Invocation is, of course, from the ring at hand and not from the Unwinder's own ring. Next, it emulates signal_search in proceeding to deal with those rings which appear in the Gatekeeper's <rtn_stk> (see BD.9.01) as pending returns. Say returns to rings i, j, k and i are as-yet unsatisfied on the <rtn_stk>; the Unwinder will then invoke, from the appropriate rings, any procedures established with appropriate invocation number as handlers for cleanup in <signals_i>, <signals_j>, <signals_k>, and <signals_i> again. There is a complication, however, in that it is not intended to "unwind past lbl". That is, the unwinding process may be looked upon as a progressing through the Stack frames of those procedures which will not be returned to at all, neither by a normal return sequence nor by an abnormal return, executing procedures on the cleanup list with the ring number, Stack frame, and invocation number of the procedures being circumvented and freeing their Stack frames. (The "abnormal return" may, for that matter, be viewed as a short-circuiting of the normal return "circuit".) At some point in this processing, however, the Stack frame of the procedure which contains the label (lbl) to which the abnormal return is being taken will be encountered, and at this point unwinding must cease. So the Unwinder actually checks each Stack frame it encounters before processing the frame (there may be many such frames for a given invocation number - which is to say, for a given period of residence in the protection ring at hand of the Multics-sense "process" which invoked the Unwinder): if the Stack frame corresponds to the frame of lbl, the Unwinder proceeds directly to its final task, which is to effect the abnormal return; otherwise, it continues to "unwind".

Note, by the way, that PL/I "non-local go to's" are not treated as direct transfers; rather, they are abnormal returns from a subroutine to a point other than where it was called from - such returns being effected by an intermediary subroutine, the Unwinder. Be it further noted that Multics compilers must not compile direct transfers to non-local labels, but must instead compile a suitable call to the Unwinder for something like "go to error;" when error is external or a parameter.

Essentially, the foregoing discussion has dealt only with the solution to the "unfinished business" problem of abnormal returns. Although invocation of cleanup handlers from the rings they were established in is sound protection technique, the basic problem of assuring protection for the abnormal return to lbl itself remains unsolved thus far. The solution lies in the method of performing the return to lbl.

A single step in the logic of the Unwinder serves both to assure the validity of the proposed abnormal return and to prevent any attempted circumvention of the protection mechanism by a direct transfer. That step is a call to the get_ring entry of the Basic File System (BG.3.01) - the same entry used by the Gatekeeper (BD.9.01) to verify gates and to get target-ring numbers. The Unwinder, however, is not interested in whether lbl is a "gate"; indeed, lbl must not be a gate, but a "door". BD.9.00 contains further discussion of gates and doors, but for present purposes it is sufficient to observe the following: If the Gatekeeper were invoked on a wall-crossing fault and determined that the faulting instruction had been a transfer, it would treat the situation as a call. Then, if the "call" were inward, the Gatekeeper would invoke get_ring to verify the target address. Get_ring examines the "protection list" (see also BG.9.00, BX.8.02) of the segment containing the target and indicates whether or not the target is a gate. Here is the crux of the matter, for if abnormal return labels were treated simply as gates, the Gatekeeper could be tricked into passing a direct transfer to such a label, as if it were a call. But such labels are not "gates", they are "doors", and are indicated as such in protection lists. Therefore, the Gatekeeper would reject a direct transfer to a label (accepting, of course, call-sequence transfers to legitimate entry points) on the evidence furnished by get_ring. This is as it should be, from the stand point of protection, because havoc could result if control were allowed to pass to an abnormal return point of an inner ring without preparations having been made in terms of stack pointers and frames, linkage pointers, and the like. Hence, the System enforces the rule that the Unwinder must be used for abnormal returns to inner rings. (Intra-ring abnormal returns and abnormal returns to outer rings also should be performed by Unwinder; this is impossible to enforce, however - and unnecessary to enforce, for any chaos resulting from failure to "unwind" stack frames and the like will only reign in the kingdom of the user who refused to allow the protection mechanism to protect him from himself.) On the other hand, once duly invoked, the Unwinder must also contribute to preserving the integrity of the environment in which an inner ring procedure will find itself when control returns to it at an abnormal return point. Therefore, it will reject any target points which are entry points, and will only perform its "unwinding" when invoked to effect an abnormal return to an abnormal return label, or "door". The Unwinder may, for that matter, be thought of as playing "doorkeeper" to the Gatekeeper's gatekeeper.

Once verification of the abnormal return has been obtained - after "unwinding" as discussed above, of course - the Unwinder can exercise its authority as a part of the protection mechanism and by subtle modification of Stacks and the Gatekeeper's <rtn_stk> cause the abnormal return to take place in such a way that control ends up in the proper ring. Details of this rather tricky undertaking are given in the discussion of Implementation, below. For now, it is enough merely to claim that the Unwinder does, indeed, solve the problems of "unfinished business" and of protection in the event of abnormal returns.

A final general point on the abnormal return mechanism: procedures which have been placed on a cleanup list by calls to condition may of course be removed by calls to reversion if it has been determined by a procedure that its "unfinished business" no longer needs to be transacted. One can conceive of cases, indeed, where this sort of thing must be done. For example, consider a PL/I epilogue: on entry to a block, the epilogue might be placed on the cleanup list to guard against an abnormal return out of the block; if no abnormal return occurs, then immediately before invoking the epilogue at the end of the block the procedure must remove the entry from the cleanup list to prevent the epilogue's being extraneously invoked in the event of an abnormal return from a subsequent block.

Error Handling

There are three error conditions which the Unwinder could encounter which could be of interest to the user: 1) The inter-ring label to which the abnormal return is being attempted is not a "door". 2) The inter-ring label to which the abnormal return is being attempted is a door but does not correspond to any stack frame encountered while "unwinding" through the return stack (<rtn_stk>); that is, the label was not passed by a procedure which has a pending return active. 3) The intra-ring label to which the abnormal return is being attempted does not correspond to any stack frame encountered while "unwinding". The Unwinder's general treatment of the conditions is the same; but, as will be seen, the implications and certain details are different. The general approach is to place an appropriate comment in the user's error file via a call to seterr (see BY.11.01), then call signal (BD.9.04). This is, of course, in keeping with the Multics error policy as enunciated in BY.11. However, these errors require special treatment in the area of how to continue after they arise. In all cases, default handling must be as system-defined conditions. That is, the system-wide default handler for user-defined conditions (unclaimed_signal) cannot be used as the fault handler for Unwinder errors. The problem is that the standard handler for unclaimed_signal will ask the user if he wishes to continue, and continuing is meaningless in these situations (unless prepared for explicitly; see below). Therefore, the default handler for the Unwinder errors is unclaimed_signal$anchor, which bypasses the continuation question and simply notes the error and transfers to the anchor point in the Shell (i.e., "aborts"). This solves the problem for the cases in which the signal goes unclaimed. (Actually, the default definition is a relatively easy matter: only the ring-0 signal vector need be pre-set for unclaimed_signal$anchor as the default handler, as the Unwinder calls signal from ring 0 and takes its default handler from <signals_0> unless there is an active handler elsewhere.)

It is possible for the user to have established a handler for the Unwinder's error conditions. In such cases, the Unwinder must be prepared for a return from its call to signal. At this point, the handling of the error conditions differs. In case 1), where the inter-ring label was not a door, it is safe to return to the caller, for no "unwinding" will have been performed. (The assumption is that the user's condition handler will have repaired the situation in the procedure which called unwinder.) However, in cases 2) and 3), where the address portion of the label corresponds to a door or is by definition legal because in the caller's ring, but the Stack frame pointer portion is not found on the return stack, it is not safe to return to the caller, because unwinding will have been performed. The assumption here is that the user's condition handler can not have repaired the situation in the calling procedure, for the situation has been irreparably altered by the Unwinding already performed. Indeed, case 2 is an easy mistake to make - for example, it would arise if a label were stored in "static" in EPL and the Stack frame portion of the label corresponded to a procedure which had already been returned to; however, we signal anyway because the original call to unwinder may have been intended to accomplish a modified abort - that is, one which does not go to the Shell.) Whatever the reason for such strange user behavior, in the event of a return from signal in cases 2), and 3), the Unwinder calls unclaimed_signal$anchor directly, as the calling procedure cannot continue.

Implementation

For the purpose of this discussion suppose A called B called C called D called E called F called G (where procedures A, B, E, F, and G are in ring i and C and D are in ring j; see Figure 1) and G wishes to make an abnormal return to abn. G calls unwinder, which is in ring 0, with abn as an argument,

```
     call unwinder (abn);

     dcl abn label;
```

Abn, being label data, consists of two parts; abnloc, the location of the abnormal return point, and abnsp, the current Stack frame at the time abn was defined.

The task of the Unwinder is to search backward along the path of control (i.e., G-F-E-D-C-B-A) looking for the stack frame abnsp. In the course of this search it executes cleanup routines, "undoes" ring crossings, and releases Stack frames. For the purpose of searching for abnsp and executing the cleanup routines it calls on a helper in each ring, procedure helper_n in ring n. Figure 1 shows a diagram of the Stack frames in rings i, j, and 0 after helper_i has been called by the Unwinder. Figure 2 shows the Stacks after the call to the Unwinder and the contents of <rtn_stk> for the three ring crossings involved. The frames marked dummy are the dummy frames which are inserted by the Gatekeeper when a wall crossing takes place. The helper works down the Stack executing the cleanup routines deposited in <signals_n> during execution of the procedure associated with each frame until either a dummy frame is encountered or frame abnsp is found. In either case it returns to the Unwinder. If the helper found abnsp, the Unwinder releases all frames down to (but not including) abnsp and returns to abnloc. If the helper found a ring-crossing dummy instead, the Unwinder simulates the ring-crossing, resulting in the state diagrammed in Figure 2. After the simulated ring-crossing the Unwinder calls the helper in the new ring. Figure 1 shows the Stack threading (sp|16 and sp|18) as single or double headed arrows along the side of the stacks. In addition it shows the backward cross-ring pointers which are in sp|28 of the ring-crossing dummies.

There are, then, two routines which need to be described: unwinder, which is a ring-0 procedure, and helper_i (all of the helpers helper_0,..., helper_63 are identical, however helper_i operates only in ring i). We consider helper_i first, as an understanding of the actions of unwinder is dependent upon an understanding of the actions of helper_i.

The Helper

Helper_i is called (by the Unwinder),

```
     call helper_i (abnsp, flag, lastsp, inv)

     dcl (abnsp, lastsp) ptr, (flag, inv) fixed bin (17);
```

where,

abnsp   is the stack frame that helper_i is to search for

flag   on return = 0 if a dummy frame is found before
          abnsp is found
          = 1 if abnsp is found
          = 2 if an error occurs in helper_i

lastsp   is a pointer to the dummy frame if abnsp was not found

inv   is the invocation number of that portion of <stack_i> in which helper_i is to search for abnsp.

Upon entry, helper_i initializes itself by setting cursp to point to the second frame before its own (e.g., spq in Figure 1). In addition, it obtains pointers (by calls to generate_ptr) to <signals_i>|0 (call it sig) and <signals_i>⌉[cleanup] (call it cup). The following steps are then repeated until one of the termination conditions is satisfied.

1. Have we found abnsp? If cursp = abnsp, set flag equal to 1 and return.
2. Execute any pending cleanup routines for this frame. (Note: this step is skipped if generate_ptr returned a null pointer.) For this purpose the following steps are repeated until all applicable cleanup routines have been executed.

    a. Examine the first item on cleanup stack. cup contains a relative pointer (call it fst) to the first item on the cleanup stack, i.e., sig⌉fst is the first item. It has the format,

sig|fst →	pointer to 2nd item	
+1	invocation # at time this item was added	itminv
+2	sp at time this item was added	itmsp
+4	cleanup routine entry data	cupntry

If both _itminv_ = _inv_ and _itmsp_ = _cursp_ (that is, if the invocation number and the Stack frame are appropriate) then this item is a cleanup routine which should now be executed. If either equality fails, go to step 3.

b. Set up to call _cupntry_. The call is to be made with zero arguments; however, if the sp portion of the _cupntry_ entry data is non-zero, the argument list must be supplemented in the proper manner (see BD.7.02).

c. Call reversion ("cleanup") to remove the _cupntry_ item from the _cleanup_ stack.

d. Finally, call _cupntry_. When it returns go to step 2a.

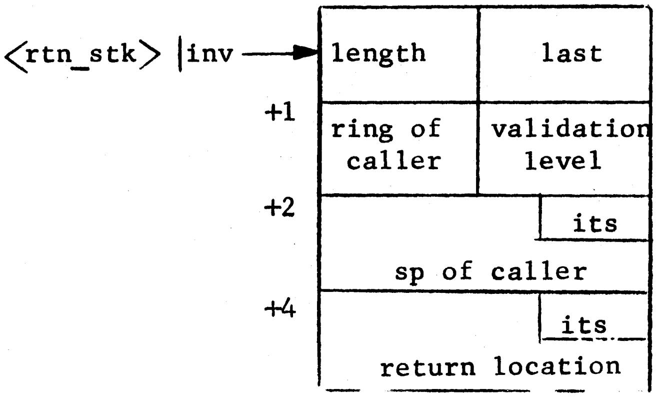
3. All cleanup routines for the current frame have now been executed. If the op field of cursp|16 = 1 this frame is a dummy which resulted from a ring crossing at this point. In this case, set _flag_ equal to 0, _lastsp_ equal to _cursp_, and return to _unwinder_.

4. Otherwise, cursp|16 should point to the previous frame. If cursp|16 ≥ _cursp_ set _flag_ equal to 2 and return. Update _cursp_ to equal the contents of cursp|16 and go to step 1.

Unwinder

When _unwinder_ is entered the following initializing steps are executed.

a. Verify the target label, _abn_. First obtain the ring of the caller. If sp|16 points back to a dummy frame then the ring number of the caller is in <rtn_stk>. (If sp|16 does not point to a dummy frame, then the caller is ring 0; the logic of ring 0 cases is discussed separately, below). <rtn_stk>|0 points to the last item in <rtn_stk> which has the format (where inv:=<rtn_stk>|0),



Set i equal to ring # and

```
call get_ring (abn, i, target_ring, type, err_code)
```

After this call, err_code is non-zero if abn is not accessible to the caller. If err_code is zero and type indicates that abn is a gate rather than a door, the abnormal return is also invalid. In both cases "unwinder_err" is signalled with an error code of 1. If abn is a door for the caller (or is in his ring) the target label is valid.

b. Set abnsp equal to the sp part of abn.

c. Information pertaining to the Stack frames of interest to the Unwinder is found in the second item in <rtn_stk> rather than the first. Accessing of this information is via the invocation number, inv. Initialize by,

```
inv:=righthalf (<rtn_stk>|(<rtn_stk>|0))
```

e.g., set inv equal to b2 in Figure 2.

The following loop is then entered:

1. Call helper_i (abnsp, flag, lastsp, inv).

2. Upon return, if flag = 1 then abnsp was found; go to step 8. If flag = 2 generate a terminate process fault. Otherwise, a ring crossing has been encountered and Stacks must be switched.

3. Update base of Stack being left, lastsb|0:=lastp|16.

4. Update the target Stack. Obtain a pointer (call it newsp) to the correct frame of the target Stack. This is found in <rtn_stk>|inv +2, e.g., in <rtn_stk>|b2 + 2 in Figure 2. Then set the invocation number, newsb|2, with the number of the previous invocation which is found in the right half of <rtn_stk>|inv, e.g., b3 in Figure 2. The new validation level is found in the right half of <rtn_stk>|inv+1 and is put into newsb|3.

5. Finally, the second item in <rtn_stk> is removed by,

   a) new_inv:=righthalf (<rtn_stk>|inv)

   b) righthalf (<rtn_stk>|(<rtn_stk>|0)):=new_inv

   which deletes the record of this crossing by detaching the first item from the second and attaching it to the third item (see Figures 2 and 3).

6. One task still remains. Refering to Figures 2 and 3, we have simulated a crossing from ring i to ring j by updating the bases of these Stacks and deleting the Gatekeeper's record of the original crossing from ring j to ring i. However, the control thread still remains unchanged and includes frames E, F, G in ring i. The processes frames in ring i are deleted by detaching the Unwinder's frame (and its dummy) from frame G in ring i and reattaching it to the last frame in ring j. mysp|16 contains a pointer to the dummy frame, dumsp.

   a) Update the cross ring pointer: dumsp|28:=newsp; e.g., set dumsp|28 equal to spd in Figure 3.

   b) Update the first item on <rtn_stk> so the Unwinder thinks it was called from the new ring;

```
        k:=<rtn_stk>|0

        <rtn_stk>|k+1:=<rtn_stk>|inv +1

        <rtn_stk>|k+2:=<rtn_stk>|inv +2

        <rtn_stk>|k+4:=<rtn_stk>|inv +4
```

   It should be pointed out that the above method of abandoning the processed Stack frames leaves the frames in the last invocation of each ring intact and preserves the threads (sp|16 and sp|18) within the ring. This gives the debugging routines something to work with in case the Unwinder is unable to find abnsp and aborts by a terminate process fault.

7. Finally set up for the new call to a helper

   a) get new ring #; i:=lefthalf(<rtn_stk>|inv+1>)

   b) inv:=new_inv

   Go to step 1.

8. If the helper found frame abnsp, the target has been reached and a return to abnloc must be simulated.

   a) Update the first item on <rtn_stk> so that it looks like the Unwinder was called from abn

```
k:=<rtn_stk>|0

<rtn_stk>|k+4:=abnloc

<rtn_stk>|k+2:=abnsp
```

b) Update the dummy frame so that the Unwinder will go to abnloc when it executes the return sequence.

```
dumsp|20:=abnloc

dumsp|0,...,dumsp|5:=abnsp|0,...,abnsp|5
```

The contents of dumsp|6 and dumsp|7 must remain unchanged in order for the return to function correctly.

The Unwinder now executes the standard return sequence.

We have not considered what happens if the Unwinder is called from ring 0. Helper_0 is the same as all the other helpers since its Stack frame is in the same relative position to the frame of the caller of unwinder as it is with any other helper (compare Figures 1 and 4). There are minor modifications in the initialization of the Unwinder. In step a) the ring #, i, is zero. In step c) inv:=<rtn_stk|0>.

In the main loop of the Unwinder two cases have not been considered: I) the ring being switched to is ring 0, and II) the ring being switched from is ring 0.

Case I: (see Figure 5) Steps 1 through 4 remain the same. We now remove the top two items in <rtn_stk>. In addition the dummy frame preceeding the Unwinder's frame is eliminated by absorption into the frame preceeding it. Hence we have new steps 5-7.

5'. Absorb dummy frame.

```
dumsp:= mysp|16

presp:= dumsp|16

presp|18:= dumsp|18

mysp|16:= dumsp|16
```

6'. Delete top two items in <rtn_stk>.

```
<rtn_stk>|0:= righthalf (<rtn_stk>|inv)
```

7'. Setup to call helper.

a) Get new ring #; i:=0

b) inv:= <rtn_stk>|0

Case II: (see Figure 6) Steps 1, 2, and 7 remain the same. None of the items are deleted from <rtn_stk>. All of the frames in ring 0 between the Unwinder and the dummy which helper_0 found are absorbed into the dummy.

The new versions of steps 3-6 are,

3". Nothing (the Gatekeeper will do this and the next step when helper_j is called).

4". Nothing.

5". Nothing is removed from <rtn_stk>; however, in preparation for step 7: new_inv:= righthalf (<rtn_stk>|inv).

6". Absorb frames;

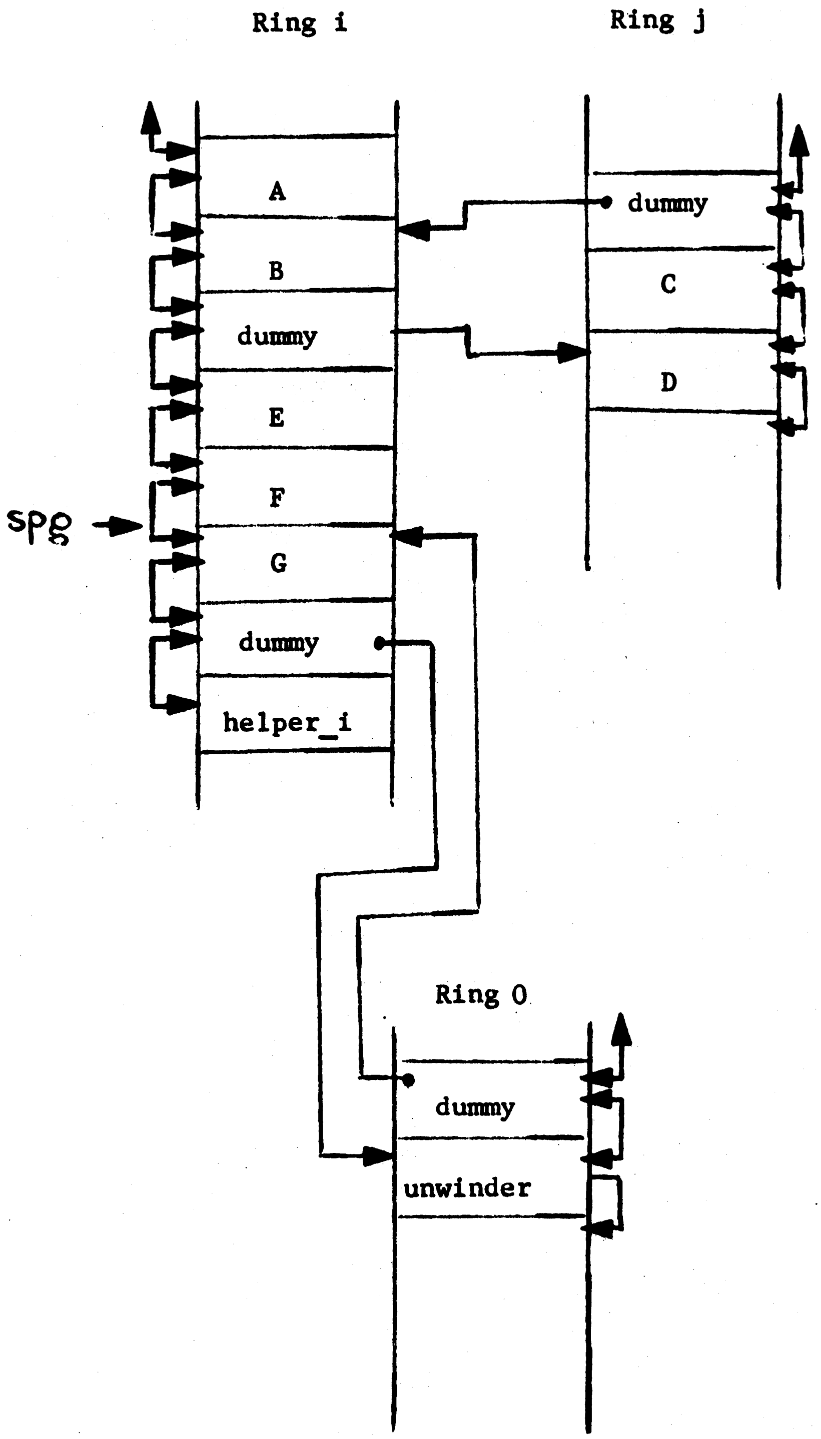
```
lastsp|18:= mysp

mysp|16:= lastsp
```

Finally, step 8 requires modification if the target, abnloc, is in ring 0. Since the Unwinder is also in ring 0, modification of <rtn_stk> is unnecessary as no ring crossing will occur when the return to abnloc is simulated.

8". Simulate return to abnloc; update the unwinder's frame,

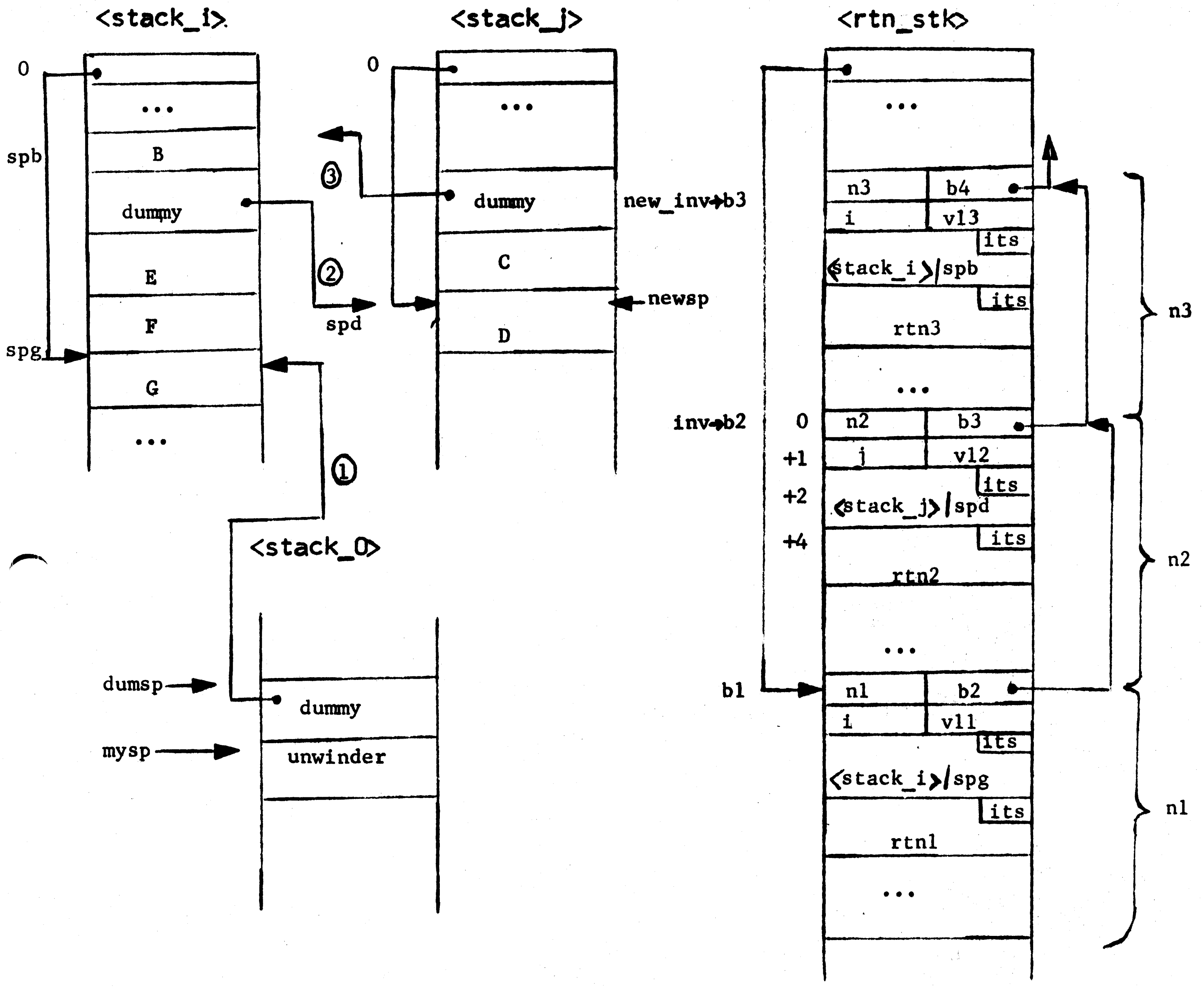
```
mysp|16:= abnsp

abnsp|20:= abnloc
```

and execute the standard return sequence.



After the unwinder has called a helper

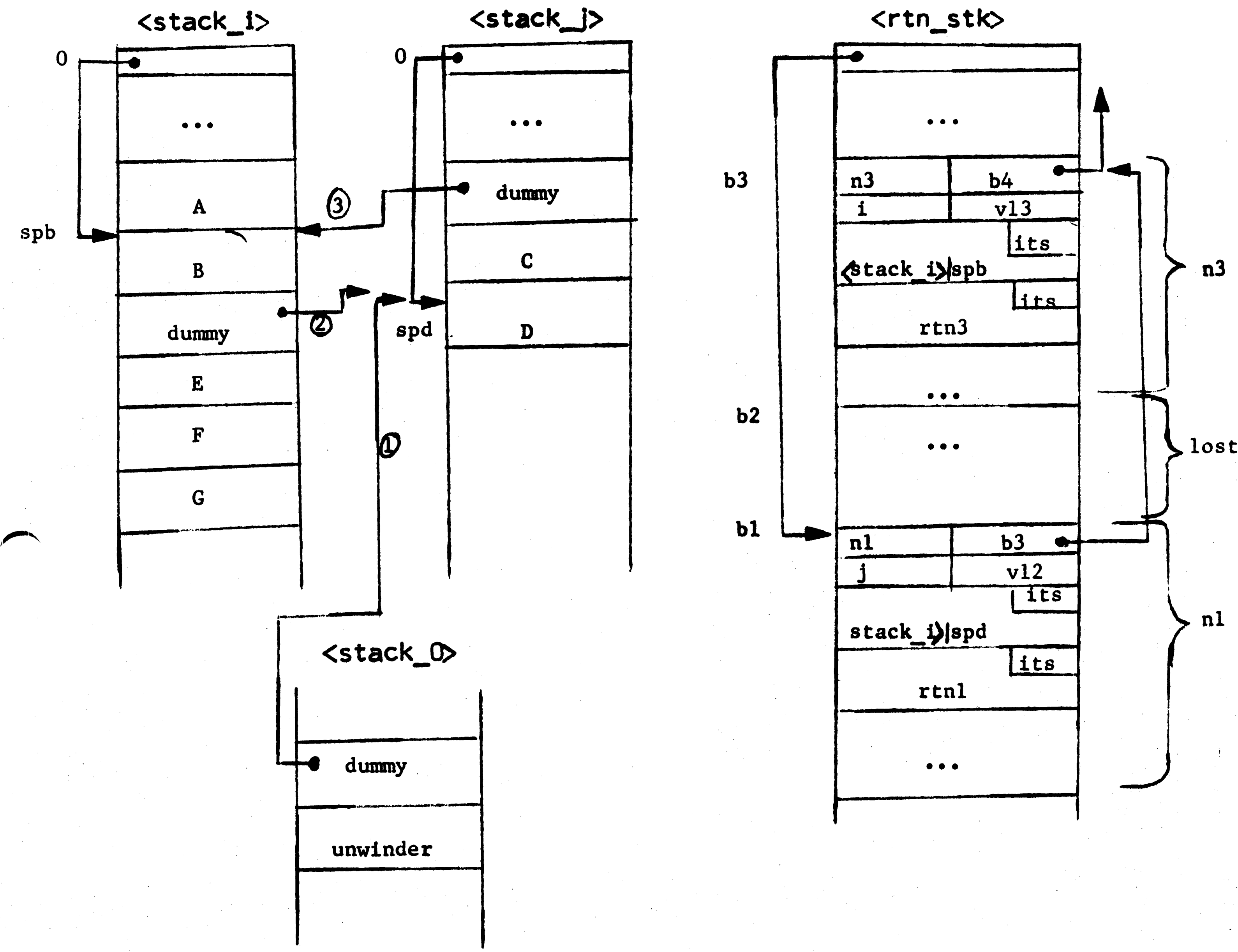
Figure 1.



Circled numbers on the left correspond to like-numbered entries in the <rtn_stk>; i.e.,

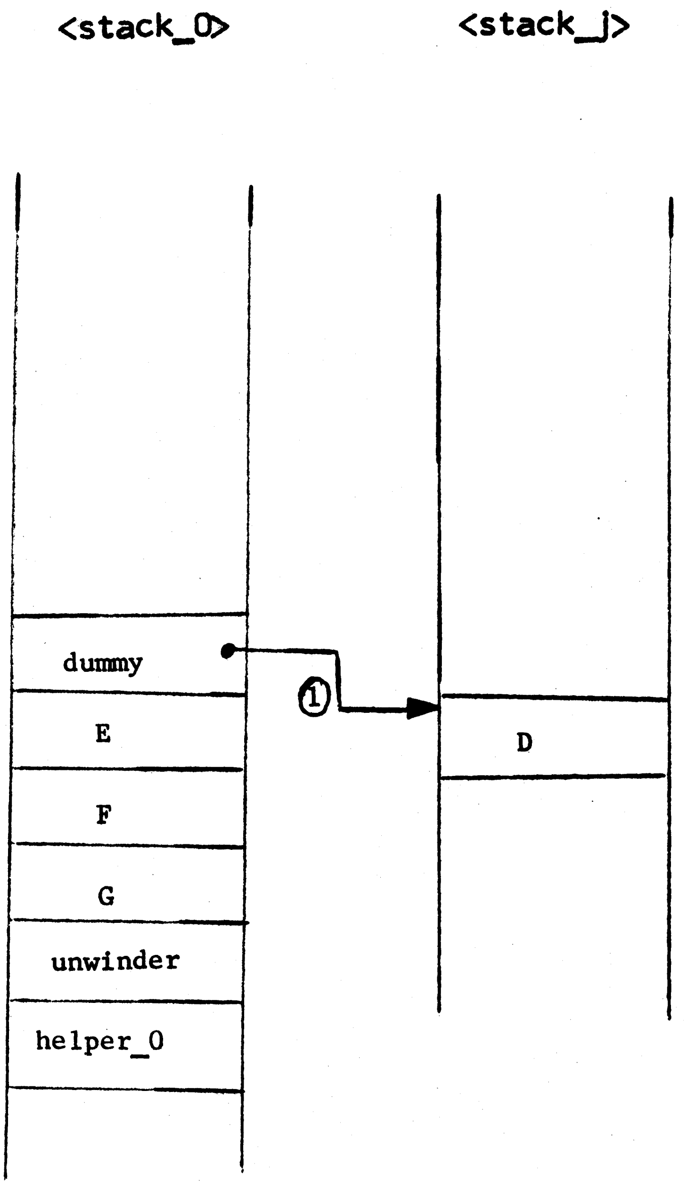
ⓘ is the ring crossing recorded in <rtn_stk>|bi before the unwinder simulates a ring crossing

Figure 2



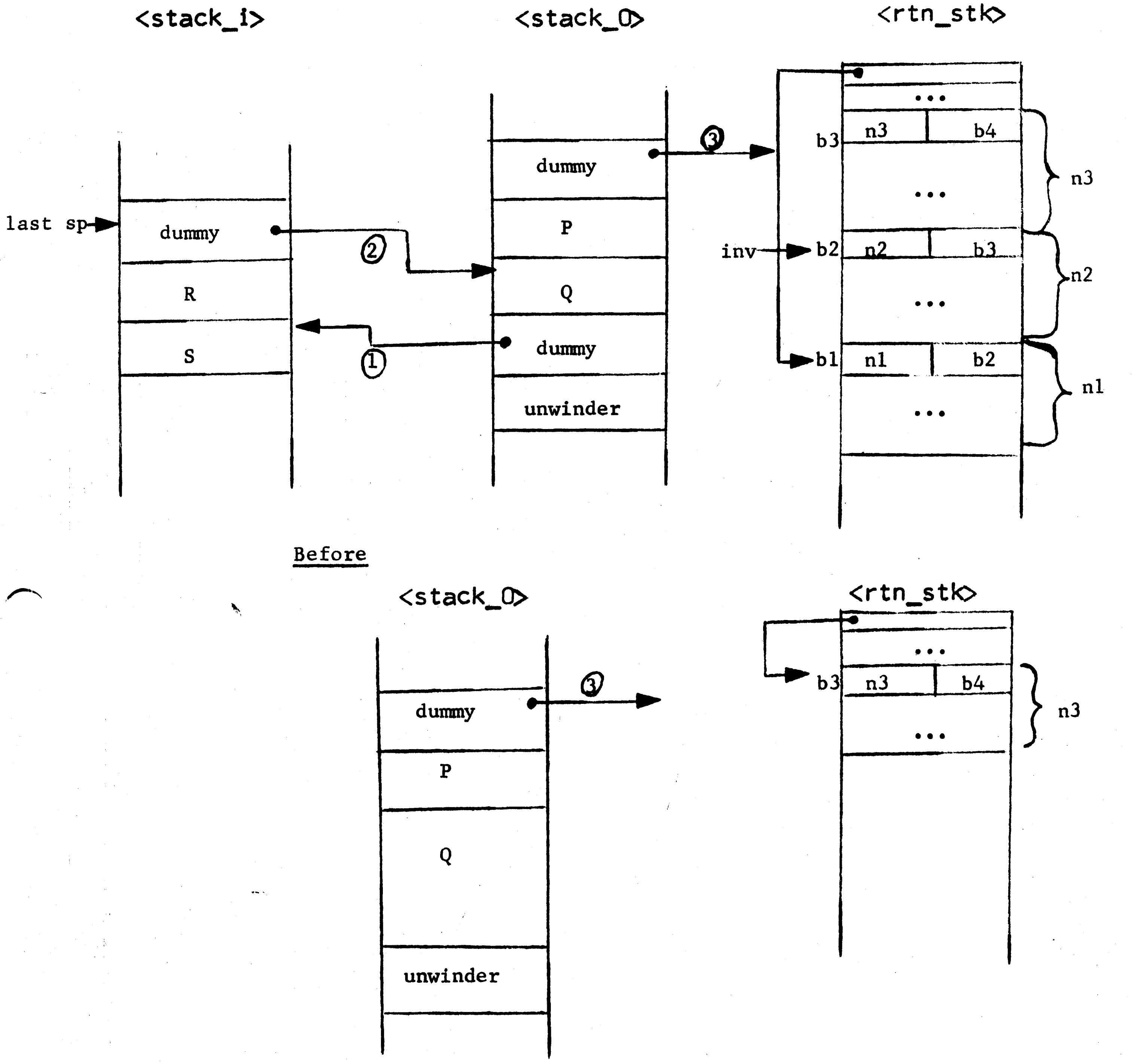
After the unwinder simulates a ring crossing

Figure 3.



The unwinder has been called from ring 0 and has called helper_0.

Figure 4.

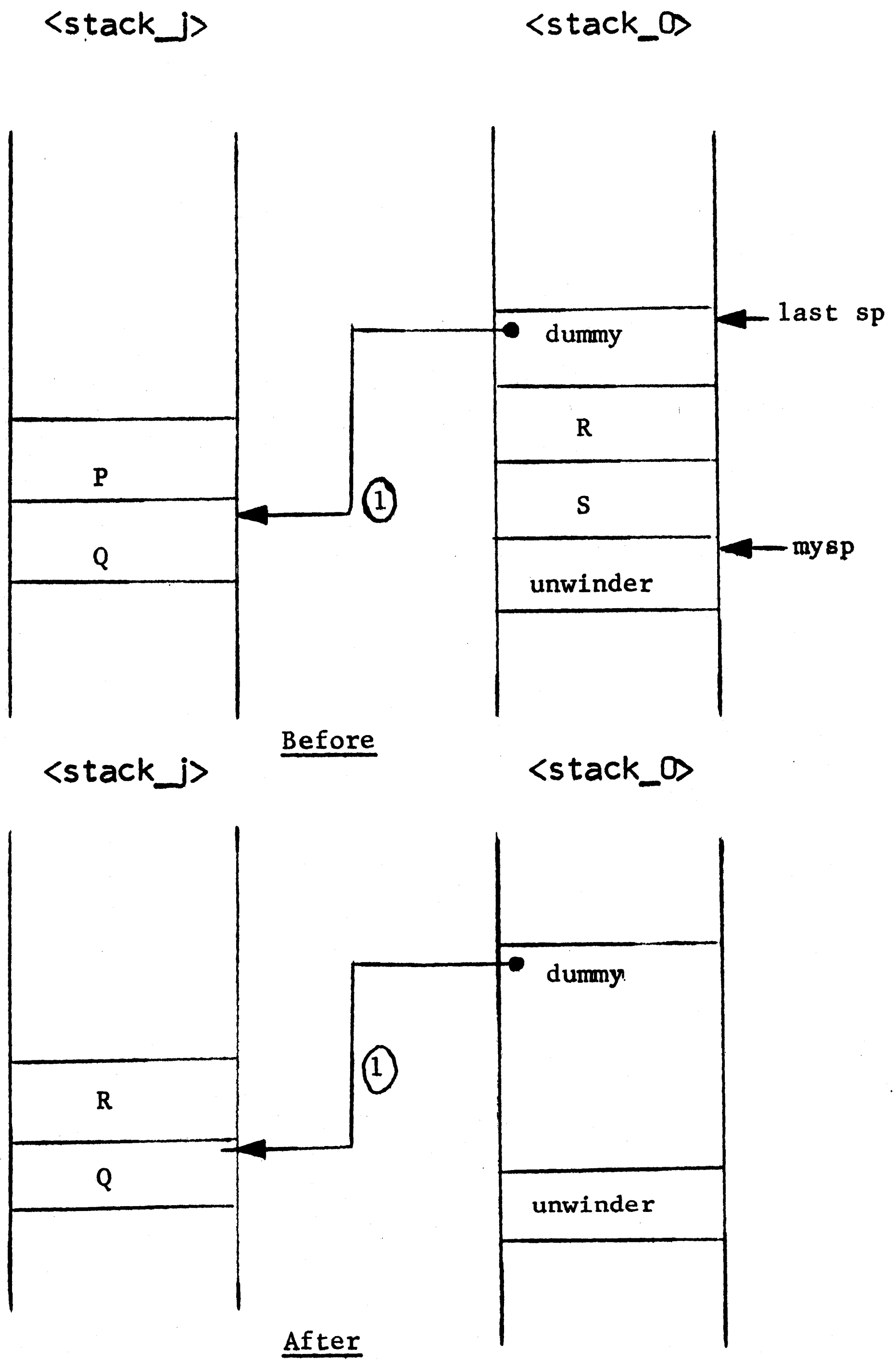


Before

After

Ring being switched to is ring 0

Figure 5.



Before

After

Ring being switched from is ring 0

Figure 6.