Published: 02/20/67

Identification

Gatekeeper: gate$in, gate$out, gate$switch
R.M. Graham, M.A. Padlipsky

Purpose

The three entries to the Gatekeeper comprise a ring-0 slave procedure which coordinates the legality checking of attempted protection wall crossings resulting from call and return sequences and handles the housekeeping and stack switching involved in such crossings. Two of the Gatekeeper's entries are called by the Fault Interceptor: gate$in is called on receipt of a Directed Fault 2, which fault results when the referenced procedure is "in" a lower-numbered protection ring than is the referencing procedure; gate$out is called on receipt of an attempt-to-execute-data fault, which fault results when the referenced procedure is "in" a higher-numbered protection ring than is the referencing procedure. (In discussions of the protection mechanism, "inward" means toward the Hard Core Supervisor's ring--ring 0--and "outward" means away from it.) In its role as fault handler for the aforementioned faults (the "protection" faults), the Gatekeeper also deals with cases where a protection fault has arisen which did not result from a call/return sequence. The third entry to the Gatekeeper, gate$switch, is called by a Hard Core Supervisor procedure which is switching protection rings without going through the fault mechanism.

Calls

The Gatekeeper's entries are called as follows:

```
call gate$in (mc_ptr, err_rtn, err_code);
call gate$out (mc_ptr, err_rtn, err_code);
call gate$switch (mc_ptr, type, ringno, err_rtn, err_code);
```

with arguments declared

```
dcl mc_ptr ptr, err_rtn label, (ringno, err_code) fixed
    bin(17), type bit (2);
```

Mc_ptr is a pointer to a copy of the faulting procedure's machine conditions (which include the 8 bases, 8 registers,

scu instruction information, and ring number). The Gatekeeper will alter these data, but cannot cause their restoration, so a copy which will subsequently be restored by the Fault Interceptor must be accessible to it. Err_rtn is the location to which the Gatekeeper will return if an attempt at an illegal crossing is detected, or if some other problem is encountered during the wall crossing. Err_code will contain a code indicating just what error was encountered. The additional arguments for gate$switch have the following meanings: Type indicates which type of crossing is desired; the coding is 00 for inward call, 01 for inward return, 10 for outward call, 11 for outward return. Ringno is the ring number of the target procedure; this is necessary only for calls.

Method

There are four cases of attempted wall crossing to be dealt with. With "inward" and "outward" defined as above, they are: inward call and its converse, outward return; and outward call and its converse, inward return. Except in the case of entry at gate$switch, the differentiation between calls and returns is based upon the nature of the faulting instruction (found in the copy of the machine conditions): Transfer class instructions imply calls, return instructions imply returns. The differentiation between inward and outward is dictated by the entry called. (In the gate$switch case, a switch is set to "1"b to indicate that certain steps in the logic which are superfluous to non-faulting calls and returns can be branched around; to minimize complication of presentation, only the main stream of the logic will be treated in the ensuing discussion.) The Gatekeeper processes each of the cases separately, as follows:

Inward Call

Figure 1 presents a block diagram of the gate$in entry to the Gatekeeper and of the inward call processing. Note that if the faulting instruction is neither a transfer nor a return gate$in rejects the attempted wall-crossing out of hand. This is a consequence of the fact that the descriptor segment for each ring is so constructed as to lead to "protection" faults on references to data segments or not according to the access information of the particular data segments involved. Hence, if a protection fault

did occur on a data reference the reference is "automatically" known to be illegal. Another way of putting it is that only procedure segments have "gates" (see Overview, and below).

1. Save return information. a) To facilitate returns and "unwinding" (see BD.9.05), certain items of information must be saved at this point. The items are the faulting procedure's stack pointer (referred to hereinafter as oldsp), ring number (oldring), and "validation level". The notion of "validation level" is discussed in BD.9.00; briefly, it may be thought of as the de jure ring number as opposed to the de facto ring number. That is, when an outer procedure calls an inner procedure it is frequently necessary that the process involved appear not to have the real ring number of the inner procedure because the latter is too highly privileged; so the inner procedure uses the ring number of its caller as its current validation level. As an example, consider the case of a linkage fault in a user's procedure: The Linker calls the Segment Housekeeping Module, which will manage the search for the segment involved. If the SHM, which is in ring 1, does not already have the segment in its Segment Name Table, it must call the Basic File System, which is in ring 0. However, this particular call to the BFS is in behalf of the user's procedure and should not be treated as a call from ring 1. Therefore, the SHM will have set the validation level to the ring number it was called from before itself calling the BFS. In general, then, the Gatekeeper must get the current value directly from its fixed location, sb|3, and insure that the existing value is not less than the value of oldring. (Outer ring procedures may themselves raise their validation levels above their ring numbers, but they must not be permitted to lower them below their ring numbers.) Oldsp and oldring are found in the copy of the machine conditions. b) The saving is done on a per-process push-down stack kept in a segment called rtn_stk. Each entry in rtn_stk is 4 words long, with oldsp in the first two words, oldring in the third, and validation level in the fourth. The index of the last (latest) entry is kept in < rtn_stk > | 0.

2. Verify access. Call the Basic File System procedure get_ring (see BG.3.01), which will check the access control information of the segment to which the transfer is being attempted and return its ring number if the transfer is legal:

```
call get_ring (address, ring, new_ring, err_rtn,
     err_code);
```

where address is a pointer to the location being transferred to, ring is the ring number of the faulting procedure, new_ring is a return argument which will be set to the ring number of the target procedure, and err_rtn is the label of an error return which get_ring will take if the crossing at hand is not legal, after indicating the type of illegality in err_code. (After determining that the ring relationships are permissible, get_ring also checks that the specific transfer at hand is directed at a legitimate entry point - or "gate". The file system maintains lists of gates for segments; see BG.9.00.) On return to error, the Gatekeeper sets err_code to 3 and transfers to err_rtn.

3. Change ring number. On normal return from get_ring, new_ring is stored into the apparent machine conditions and processing continues.

4. Switch stacks. Each protection ring has a specific stack associated with it. This is necessary in order to insure the integrity of the data belonging to more protected procedures; if less protected (and less trusted) procedures cannot even use subsequent frames of the same stack which is used by "inner" procedures, much less get at previous frames, then the opportunities for accidental (or even willful) "clobbering" are eliminated. The apparent machine conditions must be altered such that the new ring's stack will be in use after the Fault Interceptor restores machine conditions. This is a rather delicate process, involving the following ordered steps (refer also to Figure 2):

   a. Get new stack frame pointer. By convention, the zeroth location of a stack segment contains a pointer to the last used "frame" in the stack, which frame, in turn, contains a pointer to the next empty frame. (See BD.7.00 for a discussion of stack frames.) To access this information, however, it is necessary first to determine where the base of the new stack actually is. An ordered list of such locations is kept in the process definitions (pdf) segment, starting from symbol stacks. (See BJ.1.06 for a discussion of the pdf.) Hence, the pointer to the base of new_ring's stack is accessible at

```
<pdf>|[stacks]+2*new_ring.
```

   If this pointer is null, the stack segment for new_ring

does not yet exist. In this case, call appendb and estblseg (BG.8.02) to create the new segment; then initialize it. (See Appendix A, for details of stack creation). Call this segment new_stack. Then, <new_stack>|0 points to "last_used", and last_used+18 by definition points to the next empty frame in the new stack. Call the beginning of that frame newsp.

b. Update stack bases. On leaving a ring and its stack, it is necessary to update the last_used pointer at the stack's base; therefore, <old_stack>|0 is set to point to oldsp. On entering a stack, the needs of the signalling mechanism (see BD.9.04) dictate the updating of the "invocation number" which is stored immediately after last_used at the base of each ring's stack; therefore, <new_stack>|2 is set to the current value of the rtn_stk index (contents of <rtn_stk>|0). Also, <new_stack>|3 must be set to the same validation level value which was saved in step 1a.

c. Set up new stack frame. So that the eventual return from the called procedure can return to the proper stack at the proper place (i.e., to <old_stack> at oldsp) and so that processing can continue in the new stack as if the call had not come from a different stack, the frame in new_stack must be carefully fabricated. First, copy all of the first 32 words of the calling ring's stack frame (i.e., from newsp to newsp+31). The bases, registers, and return information is thus preserved.

At this point it is necessary to verify the legitimacy of the planned return. The faulting procedure's segment number, reflected in the copy of the machine conditions furnished, is checked against the segment number of the return location which is pointed to by sp|20 in the faulting procedure's stack. If the numbers are not the same, an error condition exists: set err_code to 2 and transfer to err_rtn. If the numbers do agree, the copied frame (in new_stack) is secure from the possibility of accidental tampering which could have resulted from the fact that old_stack, being a segment, is shareable. That is, it is in principle possible (though in practice quite a delicate undertaking) for process A and process B both to be using old_stack: then if a procedure of A were to make a legitimate inward call and B were to become the running process after an interruption of the Gatekeeper's processing of that call, the information at oldsp|20 might have been altered any way at all by the time A became the running process again. Hence,

checking the return from the copy in new_stack assures that the return is not only preserved, but rightfully preserved.)

However, the last-and-next-frame information which was copied into new_stack is only good for old_stack. Therefore, newsp+16 must be set to point to new_stack's last used frame (the "last_used" of 4a.), and newsp+18 must be set to point to newsp+32 which is known to be the beginning of the next available frame in new_stack (because the frame we are fabricating is 32 words long).

d. Set cross-ring flag. The perceptive reader will have noted that a certain amount of mendacity has been introduced, specifically at newsp+16. The last stack frame the process "saw" was at oldsp, to be sure; however, the last frame "seen" on a per-ring per-stack basis is the one at last_used, and it turns out to be important to such programs as the debugging aids to be able to trace through the frames of a given stack in sequence. (Provision for frame-tracing is not required by the logic of the Gatekeeper.) Therefore, in order not to break the chaining of stack frames in new_stack, newsp+16 was set to point to last_used instead of to oldsp. (Oldsp is known to the Gatekeeper via <rtn_stk>, so the return to old_stack is not jeopardized.) Of course, the chaining within new_stack is not intended to cover cross-ring/cross-stack cases, but merely to allow successive frames to be inspected without introducing breaks in the chain; therefore, some indication must be given that the stack frame at hand is merely a place holder in the chain (a freak link, perhaps). The means chosen to indicate the cross-ring nature of a stack frame is the placing of a 1 in the op code field of the last-frame pointer location, newsp+16 (the op code field is, of course, ignored by the hardware when the pointer is used). This step is duly taken. To allow for the possibility of a debugger which is sufficiently privileged (and interested) to inspect stacks in other rings, cross-ring/cross-stack tracing is provided for by setting newsp+28 to point to oldsp.

e. Set sp. The stack pointer in the apparent machine conditions is set to point to the frame just manufactured in the target ring's stack; that is, sp=newsp. (The called procedure will, on entry, "climb the stack"--i.e., set up a new frame relative to newsp, according to

the next-frame pointer at newsp+18. Call this frame newersp. Then newersp+16 will be set to point to newsp, newersp+16 being the last-frame pointer. Hence, when the called procedure effects its return it will find the information fabricated in step 4c. and will wind up--after some untangling performed by the Gatekeeper's outward return processing--in the stack of the calling procedure.)

5. Report change. The Basic File System needs to be informed of all ring changes to rings other than ring 0 so that it can arrange to wire down the new ring's descriptor segment (ring 0's descriptor is always wired down, so long as the process is active). Therefore

```
call setup_ring(new_ring,err_rtn,err_code);
```

where the arguments are all as previously defined. See BG.3.05.

6. Return to the Fault Interceptor. The call may now be completed. The Fault Interceptor will restore machine conditions from the copy altered by the Gatekeeper, after further altering them to prevent the reoccurence of the Directed Fault 2.

Outward Return

Figure 3 presents a block diagram of the gate$out entry and the Gatekeeper's outward return processing. Gate$out filters transfers and returns in the same fashion as does gate$in. Note that the outward return case is the converse of the inward call. That is, after an inward call has been effected and the inner ring procedure encounters a return instruction, an attempt-to-execute-data fault will cause the Fault Interceptor to call gate$out, and the fact that a return instruction rather than a transfer instruction has faulted will cause the Gatekeeper to perform its outward return processing.

1. Retrieve return information. The current entry in <rtn_stack> is pulled off the list and the list popped up one level (i.e., decrement the index to the current entry which is kept at <rtn_stk>|0). Recall that the entry contains the stack pointer, ring number, and validation level of the (previously inward-calling) procedure being returned to.

2. Change ring number. The ring number in the apparent

machine conditions is set to the value which was saved in <rtn_stk>.

3. Switch stacks. The process of stack switching described above must be reversed at this point. The discussion uses the terminology of Figure 2, with "new" and "old" relative to the original switching, that is, "new_stack" is being left, "old_stack" is being entered.

   a. Update new_stack. The fabricated frame in the inner ring's stack (at newsp in Figure 2) must be released. This is accomplished by setting its zeroth location to point to the contents of newsp|16, for that location will be used to point to the last used frame the next time the stack is used. (Newsp comes from the copy of the machine conditions; the base of the stack being entered comes from <pdf>, in like manner to 4a., above.)

   b. Set up old_stack frame. The value of oldsp was stored in <rtn_stk>; it is now placed into the apparent machine conditions as the stack pointer, so that the proper stack and frame will be used when the machine conditions are restored.

   c. Update invocation number. As mentioned in 4b above, when entering a ring it is necessary to update the invocation number stored at word 2 of its stack. Therefore, <rtn_stk>|0 is stored into <old_stack>|2 at this point.

4. Restore validation level. The saved validation level is restored into sb|3.

5. Report change. Call setup_ring, as in 5. above.

6. Return to Fault Interceptor. The return may now be completed. The Fault Interceptor will restore machine conditions from the copy altered by the Gatekeeper, after further altering them to prevent reoccurrence of the attempt-to-execute-data fault.

Outward Call

Figure 4 presents a block diagram of the Gatekeeper's outward call processing. The outward call is a more sensitive proposition than the two cases discussed previously, as

particular care must be taken to make available the data which belong to the more protected inner ring and to protect them from unauthorized tampering by the less protected outer ring procedure which is being called. To this end, an additional step is performed in the outward call processing which is not needed for inward calls. (Otherwise, the logic is essentially the same for the two cases.) After making the standard stack switching preparations, the outward call logic dictates a call to a routine named arg_pull. The routine is discussed in detail in section BD.9.02; briefly, it takes as arguments the faulting (inner) procedure's argument pointer and the called (outer) procedure's stack pointer and copies the indicated arguments into the "new" stack. It should be noted here that arg_pull requires the appearance of data descriptions (see BD.7.02) in the argument list; this appearance is assured by use of the "callback" option in the compilation or assembly of the calling procedure (see, e.g., BP.0.00). After the arguments have successfully been pulled through the protection wall, the outward call processing parallels the inward call processing in appropriately altering the apparent machine conditions and returning to the Fault Interceptor. One small, but crucial, difference in the two cases is that the outward call must alter the apparent argument pointer to the proper value in the new stack, so that when the machine conditions are restored by the Fault Interceptor the copied, accessible arguments will be employed by the called procedure. This tactic was unnecessary in the inward call case because the arguments are accessible when they are in an outer ring.

Inward Return

Figure 5 presents a block diagram of the Gatekeeper's inward return processing. Being the converse of the outward call, this case, too, is sensitive to argument security. The logic parallels that of the outward return, with two additional steps. The additions involve, of course, undoing the corresponding steps taken in outward calls. After the standard stack switching preparations have been made a call must be made to a routine named arg_push. This routine is discussed in detail in BD.9.02; briefly, it takes as arguments the faulting procedure's argument pointer and the target procedure's stack pointer and copies into the appropriate members of the target procedure's argument list the return arguments in the faulting procedure. (Only return arguments, which have their "set bit" on, are dealt with, so that any possible tampering with other inner ring data is automatically ignored.) Finally, the other deviation from the Outward return processing is handled: The argument pointer in the apparent machine conditions is restored to its old (pre-call value.

Appendix A. Stack Creation

As even a casual reader of the MSPM will have noticed, there are many "stacks" in Multics. All too frequently, however, references are found simply to "the stack"; even the current section is not innocent in this respect. For the records, then, "the stack" in this context is meant to refer to that (call-save-return) stack used by the standard, user- procedure call, save, and return sequences. It is, in otherwords, that segment of which an appropriate portion must be pointed to by the stack pointer when a procedure is executing if that procedure is to be able to invoke another procedure and have control returned to it eventually. A more comprehensive discussion of the stack (or "Stack") will be found in BD.7.00. However, for present purposes it is necessary to accept as given the existence and nature of Stack, and to introduce a complication rather than a clarification: in order to implement the Multics Protection Mechanism it is necessary to have a separate Stack "for each ring" - that is, "for use by procedures executing in each protection ring "- but these Stacks must present, at least conceptually, the appearance of being a single stack from the point of view of the user (and his working process).

Now, the problems of stack switching and frame changing have been dealt with in the body of the current section. The problem of stack creation, which is not particularly germane to the main stream of the Gatekeeper's logic, is taken up here, with the preceding paragraph offered as background and some sort of implicit motivation.

The Gatekeeper is in charge of Stack creation. If, when a ring crossing is about to be effected, the process in whose behalf the Gatekeeper is operating has not entered the particular target ring before, there will be no stack in existence in the target ring for the Gatekeeper to switch to. (Unless the target ring is ring 0, which has a preinitialized stack.) Such a condition is indicated by the presence of a null pointer at <pdf>|[stacks]+2*n, where n is the ring number of the target ring. Creating a stack segment is almost straightforward, except for one consideration: segments need names. Therefore, we do hereby establish and declare a

> CONVENTION: For a given protection ring, n ($0 \leq n \leq 63$), the stack (or Stack), in the sense of the call-save-return stack of section BD.7, is named <stack_n>.

(The semantic purist may object that <Stack_n> would be more appropriate. He would be right. However, case shifts are a nuisance.)

The Gatekeeper concatenates "stack_" with new_ring (after converting the latter to character), calls the result name, and invokes the file system primitive appendb in order to have a "branch" created in the Process Directory (see also BD.6.09):

```
call appendb(dir, name, type, uid, mode, copy, max1,
     errtn, code);
```

where dir is the path name of the Process Directory, which is obtainable from an invocation of the library routine pdir (BY.2.06), name is as above, type = "0"b (indicating branch is a non-directory), mode is RW, and the other arguments have values appropriate to the meanings established for them in BG.8.02. If appendb reutrns to errtn, the Gatekeeper sets err_code to 4 and transfers to err_rtn.

If the branch is successfully appended, it is necessary to call the file system primitive estblseg, with appropriate (but not relevant to this discussion) arguments as dictated by BG.8.02.

Next, <stack_n>|0 must be set to 8, the "last-used" frame (which is the first frame in the stack, of course). <stack_n>|8+16 is set to null, as the last-frame pointer is meaningless in the first frame of a Stack. <stack_n>|8+18, the next-frame pointer of the first frame, is set to point to <stack_n>|8+32, allowing for the standard fixed length of a stack frame. We now have a Stack.

Figure 1. Gate_in and Inward Call Processing

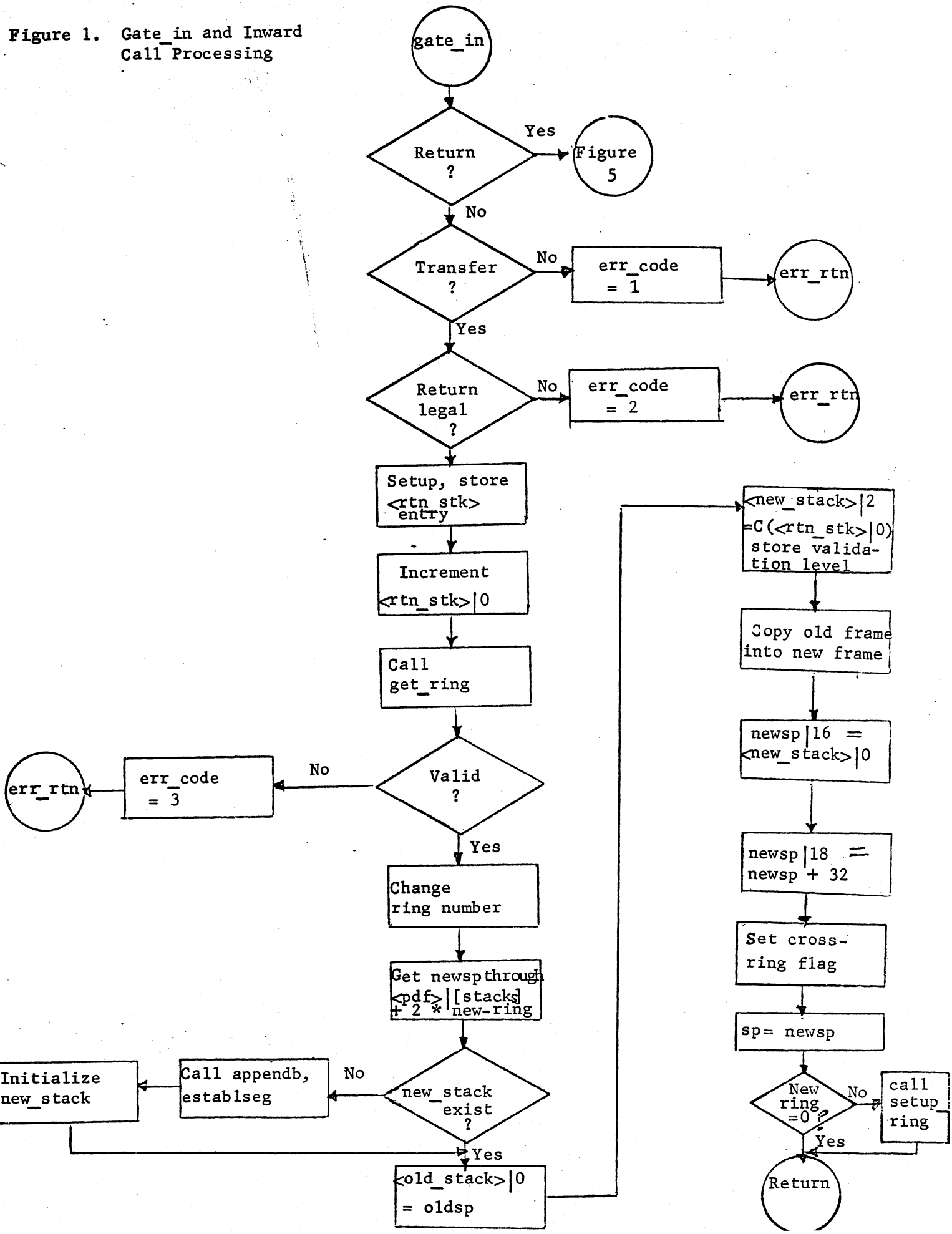
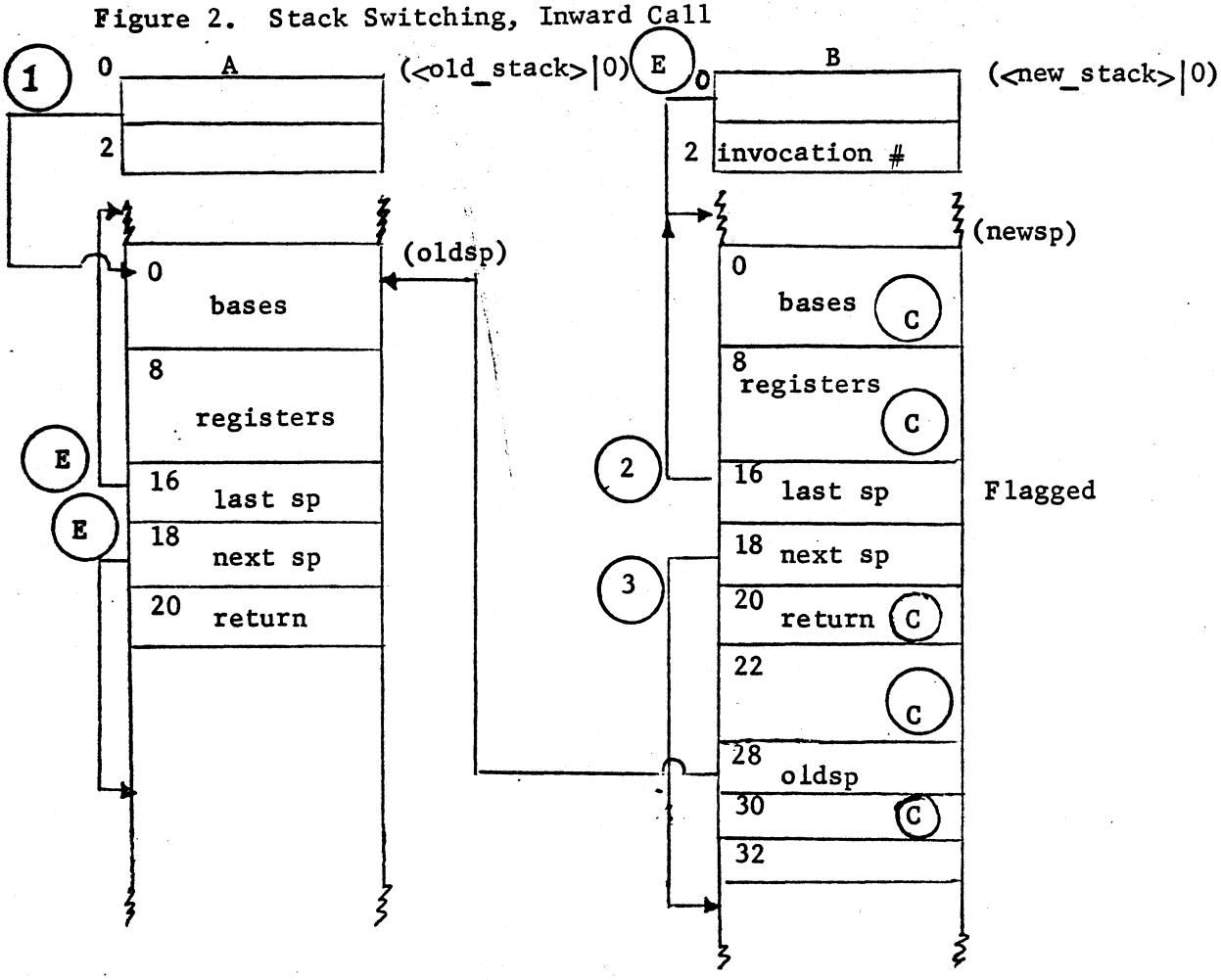


Figure 2. Stack Switching, Inward Call



Stack A belongs to the ring of the calling-procedure (old_ring), stack B to the target procedure (new_ring).

Legend: Arrows indicate pointers.

Pointers marked (E) exist prior to stack-switching.

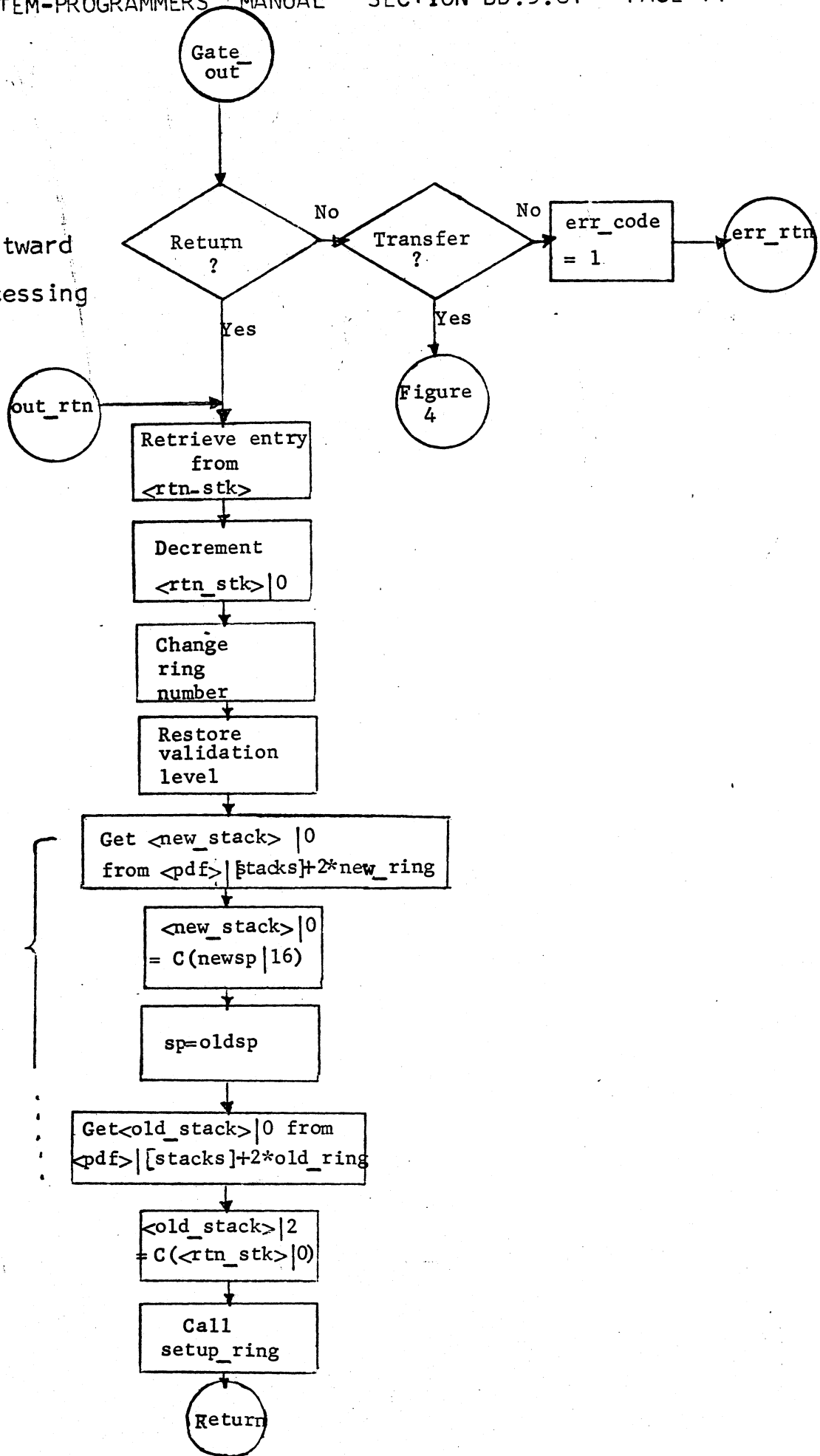
Numbered pointers are set in the order indicated by the numbers.

Areas marked (C) are copied from A to B.

Note that <new_stack>|2 receives (C <rtn_stk>|0), and that the op code field of newsp|16 is set to 1.

Figure 3.

Gate_out Outward Return Processing



"New" and "old" follow Figure 2.

Figure 4.

Outward Call Processing

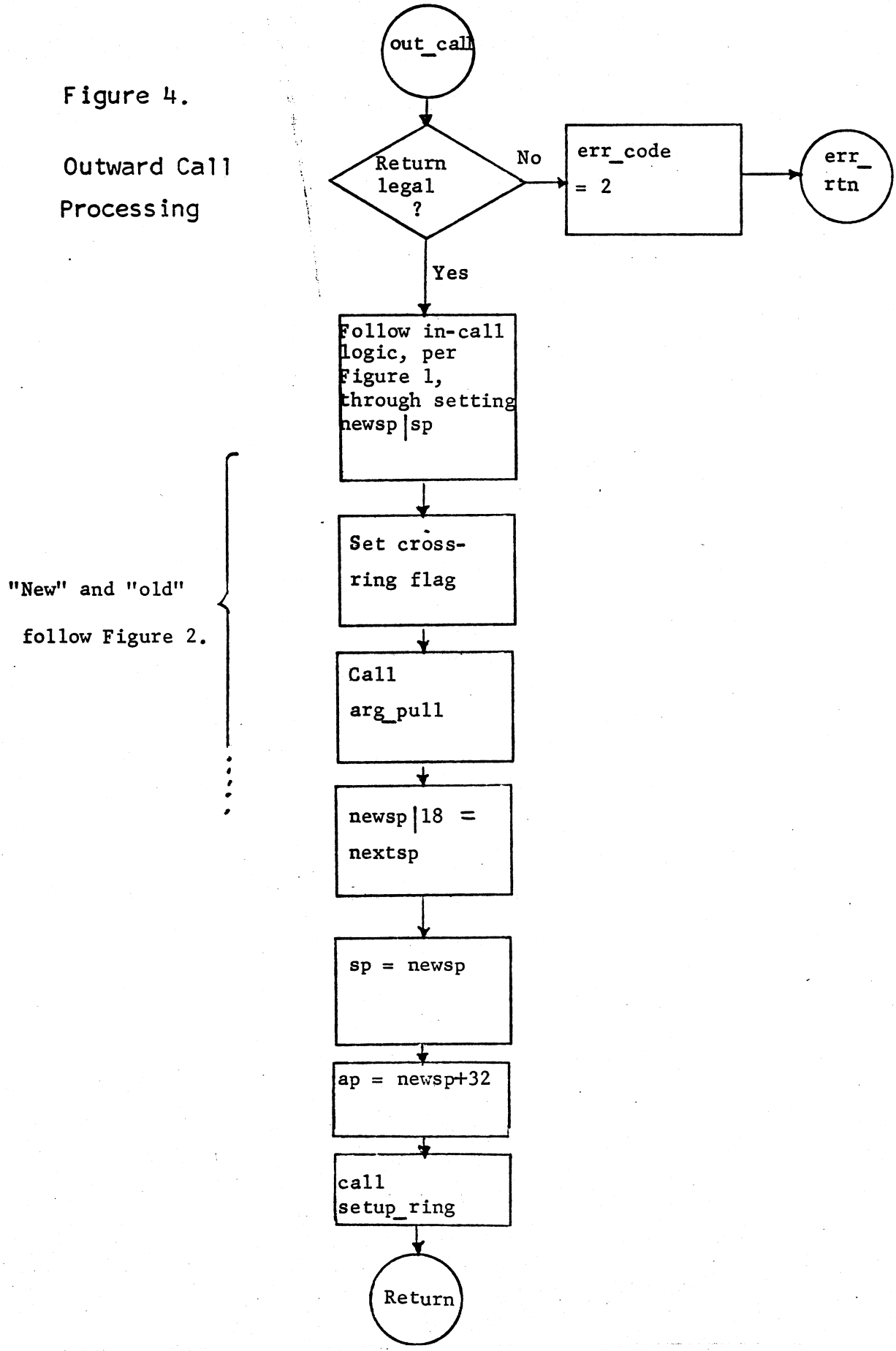


Figure 5. Inward Return Processing