MCR10138
Fix bug in ipc_$block That Can Cause Process to Loop Indefinitely
Revision 1.5

Eric Swenson

February 11, 2025

1 Important Note

This MCR revision is written after the original 1.2 review was approved and the changes described therein installed into MR12.9 (as HC 12.9a). Since that time, the Multics development team discovered that while the original fix did address the hang problem described in the next section, it causes other hangs when running the `ci-kit` integration tests. After much more troubleshooting and investigation, we determined that the original fix didn't quite go far enough in addressing the issue. Hence, we are reopening this MCR and proposing a new fix. Gary Dixon was instrumental in getting the final fix in place.

2 Problem Description

During MR12.8 testing, a user attempted to upgrade an existing MR12.7 system to MR12.8, using the provided `MR12.7_to_MR12.8_upgrade.ini` dps8m simulator script. While this script worked fine for many other users, a combination of the specific system that was being upgraded and the operator commands in the script caused the Initializer process to appear hung after a successful upgrade of the storage system to MR12.8, and after starting normal Multics service. The symptoms were that some operator commands queued in the simulator script were not getting executed, and it was not possible to enter operator commands on the operator console. While the system was running fine, the Initializer process was apparently "wedged". In fact, the Initializer process was not completely wedged because event call channel wakeups were being processed, as evidenced by act_ctl_ messages and other messages appearing on the system console.

Details as to the real cause, and the analysis that revealed the cause (and led to a fix) appear in a subsequent section. However, it may be helpful to know that the actual bug had nothing to do with MR12.8, nor with the upgrade from MR12.7 to MR12.8. Rather, the cause was due to a shortcoming in user-ring IPC, where events were not being picked up from ring 0 and processed correctly. The problem was prompted by the fact that the `admin` operator command was executed at the "wrong time"—just after the `startup` command was executed, and before the Utility.SysDaemon's `start_up.ec` had been able to have its `sac delete_old_pdds` command executed by the Initializer

process. In fact, the Initializer process was in "admin mode", and as a result of the bug in the user ring IPC mechanism, not able to process the `ame` command to exit admin mode. Furthermore, due to fact that `sac` commands are not allowed to execute in "admin mode", the "sac delete_old_pdds" command was unable to be executed by the AS. The timing of the various commands is wholly dependent on state of Multics storage system (the disks) and of any pending absentee processes that had not yet executed, but would be past due for running at the time of the next Answering Service (AS) startup.

Using the original storage system volumes that showed the above symptoms, it proved possible to reproduce the hang state by simply booting the system (using either the MR12.7 or MR12.8 Multics System Tape (MST)), as long as the Initializer `admin` command was issued directly after the `startup` command, followed by one or more `pause` commands and an `ame` command. It was not necessary to attempt an upgrade from MR12.7 to MR12.8. There appeared to be two pre-conditions causing the hang. The first is the storage system in a state where one or more absentee processes would start running as soon as the Answering Service (AS) was brought up, and the immediate entry into admin mode. For the sake of simplicity, all debugging and analysis was performed using a copy of the original storage system that exhibited the problem. However, since a second user reported seeing the exact same hang problem, it is possible that the preconditions are variable, as long as the timing of the execution of the `admin`, `sac delete_old_pdds`, and `ame` commands had to be exactly such that the `sac` command had not been processed before admin mode was entered. In fact, while investigating the cause of the hang, I simply booted the provided storage system disks (which had MR12.7 installed) using an MR12.8 Multics System Tape (MST), using a dps8m simulator script that included the `startup`, `admin`, `pause 120`, and `ame` commands – in other words, I invoked admin mode directly after bringing up the AS (`startup`).

Skipping all the details and analysis, a short summary of the underlying problem follows. This description glosses over some relevant points for the sake of brevity. Please refer to the following sections for a more accurate description of the issue.

When an process blocks waiting on an event (in this case, admin mode was waiting on a command from the operator console), it calls `ipc_$block`, which is part of user-ring IPC. `ipc_$block` looks for any wakeups on the list of event wait channels passed to it by its caller. It also looks for any wakeups on any event call channels set up by the process, invoking the corresponding handlers if a wakeup is found. The current implementation loops looking for wakeups for event call channels and the event wait channels it is blocked on, and would loop forever (as is the case in the issue at hand), as long as there are wakeups pending for any event call channels. So a process can flood itself with event call channel wakeups, preventing the process from exiting the above-mentioned loop. As long as there are unhandled event call channel wakeups, it will never exit this loop, and will never call ring 0 to retrieve additional wakeups from other processes or from ring 0. As a result, wakeups for the event wait channels on which the process is blocked that originated from non-Initializer processes will never be picked up, and the call to `ipc_$block` will never return. The process will effectively be hung (blocked and looping calling event call channels).

Now, in a normal user process, the user can exit this state by issuing a QUIT signal (an IPS signal, as opposed to an IPC wakeup). This will cause a new listener command level. However, in the Initializer process, this is not possible, and therefore the system will appear "hung" if it gets into this state.

There is one important factor that impacts the above problem: normally, when a process 1 sends a wakeup to a process 2 (via `hcs_$wakeup`), the wakeup is handled in ring 0, added to a system

table, called the Interprocess Transmission Table (ITT). When process 2 calls `hcs_$read_events` or `hcs_$fblock`, the code in ring 0 copies the event messages for that process in the ITT to process 2's Event Channel Table (ECT) (for the process 1's current ring). So if the event channel wakeup were sent to a different process than the sending process, then `ipc_$block` would never see that event until it calls either of those ring-0 gate entries. However, there is an optimization made when a process sends a wakeup to its own process. Here, the wakeup is immediately delivered to the process' ECT (for the current ring), rather than requiring the process to call into ring 0 to retrieve the wakeup. The reason why this is critical to the issue at hand, is that in the failing scenario, it is the Initializer process that is sending the flood of wakeups to itself, and thus these wakeups are immediately delivered to the ECT. Consequently, the call to `ipc_$block` in the Initializer process was able to pick up the stream of wakeups for the event call channel, preventing it from ever calling ring 0 to pick up the event wait channel wakeup for the operator console. The wakeup for operator console input results from a *terminate* interrupt arriving on some processor, and the handler for that interrupt sending the event wakeup to the Initializer process. Frequently, that interrupt would be handled by a different process than the Initializer process, and therefore be stored initially in the ITT, rather than Initializer's ECT.

The ticket describing the original problem as reported is located here: https://multics-trac.swenson.org/ticket/313.

In addition to the above issue, it was determined that a new hardcore gate entry, texttthcs_$get_assigned_channels, should be added to hardcore in order to allow user-ring programs to accurately present information about fast event channels. `hcs_` has gate entries for `assign_channel` and `delete_channel`, but no means of getting the list of assigned fast channels. An `exec_com` that displays the contents of the ECT could not accurately show which of the fast event channels had been assigned to the process in a given ring.

3 Proposed Fix

The proposed fix is comprised of two parts. The first part involves the inner loop of `ipc_$block`, which checks for event wait channels and event call channels that are already threaded into the `EV_CALL_MESSAGE` and `EV_WAIT_MESSAGE` linked lists in the ECT. That logic causes a return from `ipc_$block` if there are messages for any one of the event wait channels supplied in the event wait list parameter to `ipc_$block`, and the handling of any event call channels for which messages exist. The change is to cause this inner loop to periodically call into ring 0 to copy event channel messages from the ring-0 ITT to the current process' ECT.

The original fix (in the 12.9a hardcore) had this inner loop calling `hcs_$read_events` every N iterations will ensure that `ipc_$block` retrieves any wakeups for event channels NOT originating from within its own process. The operator console wakeups are ring-0 (device) wakeups that may not get placed directly into a process' ECT (because when they are detected by hardcore, the executing process may not be the intended recipient of the wakeup). They are placed in the ITT as mentioned previously, only to be picked up and moved to the ECT's ITT_MESSAGE list when the recipient process calls `hcs_$read_events` or `hcs_$fblock`.

We determined, however, that calling `hcs_$read_events` every N iterations is insufficient, since all this procedure does is copy the events from the ITT to the ITT_MESSAGE list in the ECT, but doesn't actually cause those messages to be threaded into either the EV_CALL_MESSAGE or EV_WAIT_MESSAGE list in the ECT. Only messages in those two lists are actually checked in the

inner loop.

The correct fix is to actually call `copy_itt_messages` periodically. That procedure calls `hcs_$read_events`, but also performs the threading, emptying out the ITT_MESSAGE list and populating the `EV_CALL_MESSAGE` and `EV_WAIT_MESSAGE` lists.

Deciding on the value of N, the number of iterations in the `ipc_$block` loop, is a necessary step in implementing the proposed fix. The value cannot be too large, or the delay before handling the event wait channel wakeups will be long. In the failing test case described above, this would mean a delay in seeing and executing the `ame` Initializer command and the exiting of admin mode. Using a value that is too small might incur more overhead and delay timely handling of event call channels, especially those sent by the same process. Event call channels are frequently used for periodic events, and may leverage timer wakeups (see timer_manager_). Calls from `ipc_$block` to ring 0 take more time (ring-crossing time and stack switching time) than calls to procedures in the current ring (such as for event call channels).

Experimentation shows that using values of 5, 10, and 20 for N work just fine to address the problem. The author recommends using a value of 5 because that lower value may provide slightly better responsiveness for queued operator commands, such as what occurs in DPS8M simulator scripts, which leverage its `autoinput` capability to queue up commands. Given that it is very unlikely that in normal usage (in any process, not just the Initializer process), there are more than 5 event call channels with pending wakeups (sent by the subject process and not another process), this choice of N seems reasonable.

The "first part" of the fix, described just above, does not completely address all the hangs that have been experienced. It is still possible to miss some wakeups. This was discovered when running the integration `ci-kit` tests repeatedly and we noticed that there were still some pauses and occasional hangs. This led us to the second part of the proposed fix.

The second part of the fix involves adjusting the logic in `copy_itt_messages`. The original logic returned immediately if it was found that there were some messages in the `ITT_MESSAGE` list in the ECT. In that case, it would not actually thread those messages into the `EV_CALL_MESSAGE` and `EV_WAIT_MESSAGE` ECT lists. And not threading them into the lists would make the caller (ipc_$block) not see the messages, and therefore not call the associated call channel handlers or cause `ipc_$block` to return if a wakeup for an awaited wait channel had been received.

The new logic determines whether there are any messages already in the ECT's `ITT_MESSAGE` list. If none is found, then it calls `hcs_$read_events` to read the events from the ring-0 ITT to the ECT's `ITT_MESSAGE` list. And then it moves those messages to the corresponding `EV_WAIT_MESSAGE` or `EV_CALL_MESSAGE` ECT list. However, if messages are found, it moves those messages to the appropriate ECT list and then makes another pass to call `hcs_$read_events` and move those messages to the appropriate lists. It returns when there are no more messages to process in the ITT.

The proposed fix solves the above hang because in the middle of processing the flood of event call channel wakeups, it will every so often call into ring 0 (`hcs_$read_events`) to retrieve new wakeups and thread them into the appropriate ECT lists. This will make available any pending wakeup for the event wait channel that is in the event channel wait list passed to ipc_$block. This wakeup will be seen by the above-mentioned loop and cause `ipc_$block` to return to its caller. This will allow, in the case of the described failing scenario, the `ame` admin command to be read and executed, exiting admin mode, and in turn allowing the AS request (`sac delete_old_pdds`) to get handled. This will stop the flood of these wakeups and the system will return to normal.

The `compare_ascii` output shows the proposed changes:

```
cpa [lpn ipc_real_.pl1] ==
Inserted in B:
B168     dcl      block_loop_count  fixed bin;
Preceding:
A164     dcl      block_val         fixed bin (3);


A750
Deleted by B, preceding:
B755                  call copy_itt_messages (NO);            /* get pending ITT messages
    */


A759     /* Check the channels for a pending event to satisfy this block */
A760
A761                  do while (check_channels = YES);
Changed by B to:
B763     /* Check the channels for a pending event to satisfy this block. Keep track of
    iterations through this loop and
B764        every 5 iterations, call into ring-0 to retrieve any additional events. This
    prevents a flood of event call
B765        channel wakeups from our own process keeping us from ever retrieving new event
    wait channel wakeups, which
B766        might allow us to exit this loop. */
B767
B768                  block_loop_count = 0;
B769                  do while (check_channels = YES);
B770                       block_loop_count = block_loop_count + 1;
B771                       if block_loop_count = 5 then do;
B772                            block_loop_count = 0;
B773                            call copy_itt_messages(NO);
B774                       end;


A934
Changed by B to:
B947     dcl      found_itt_messages bit (1) aligned;
B948     dcl      got_itt_messages  bit (1) aligned;
B949     dcl      saved_fast_events bit (36) aligned;

A938              if ect_header.firstp (ITT_MESSAGE) = null
A939              then do;
A940                        if block_val ^= cur_ring
A941                        then call cu_$level_set (cur_ring);
A942                        call hcs_$read_events (ipc_data_$fast_channel_events, ("0"b));
A943                        if block_val ^= cur_ring
A944                        then call cu_$level_set (block_val);
A945                  end;
A946
Changed by B to:
```

```
B953              found_itt_messages = (ect_header.firstp (ITT_MESSAGE) = null() );
B954              if  ^found_itt_messages then do;
B955
B956     READ_NEW_MESSAGES:
B957                   saved_fast_events = ipc_data_$fast_channel_events;
B958                   got_itt_messages = NO;                  /* Setupto determine if
    hcs_$read_events returned  */
B959                                                           /* new fast wait_channel
    events, or copied messages into */
B960                                                           /* the ITT_MESSAGE thread.
                           */
B961
B962                   if block_val ^= cur_ring
B963                        then call cu_$level_set (cur_ring);
B964
B965                   call hcs_$read_events
    (ipc_data_$fast_channel_events,got_itt_messages);
B966
B967                   if block_val ^= cur_ring
B968                        then call cu_$level_set (block_val);
B969
B970                   if  ipc_data_$fast_channel_events = saved_fast_events
B971                    &  ^got_itt_messages
B972                    &  ect_header.firstp (ITT_MESSAGE) = null() then
B973                        return;                            /* Did not find any new
    messages or fast events           */
B974                   end;
B975

Inserted in B:
B1055             if  found_itt_messages & ect_header.firstp (ITT_MESSAGE)= null() then do;
B1056                                                          /* ASSERT: the ITT_MESSAGE
    thread should always be empty */
B1057                  found_itt_messages = NO;                /*  at this point in the
    code.                      */
B1058                  goto READ_NEW_MESSAGES;
B1059                  end;
Preceding:
A1026             return;

Comparison finished: 9 differences, 70 lines.
```

In addition to the proposed changes above, a new gate entry, `hcs_$get_assigned_channels` is proposed. This ring-0 gate entry will return to the caller the fast event channels that have been assigned to the process at its current validation level. This necessitates three additional changed files.

The first change is to the `hcs_` gate itself and its change is shown below:

```
Inserted in B:
B91               hgate    get_assigned_channels,hc_ipc,get_assigned_channels,2
Preceding:
```

```
A91               hgate    get_author,status_,get_author,5,bad_dir_trap

Comparison finished: 1 difference, 1 line.
```

The change is straightforward; the new entry invokes the new hardcore entry `hc_ipc$get_assigned_channels`. The number 2 at the end of the gate entry definition signifiges that the entry takes two parameters.

The second change is to add the new entry to `hc_ipc.pl1`. That change is shown below:

```
Inserted in B:
B337     get_assigned_channels:
B338         entry (a_special_channels, a_code);
B339
B340             dcl a_special_channels bit (36) aligned;
B341             dcl temp_channels bit (36) aligned;
B342
B343             val_ring = level$get ();
B344             temp_channels = pds$special_channels & pds$event_masks (val_ring);
B345
B346             a_special_channels = temp_channels;
B347             a_code = 0;
B348             return;
B349     \014
Preceding:
A337     delete_channel:

Comparison finished: 1 difference, 13 lines.
```

This change involves getting the user's current validation level, and then computing the bitwise `AND` of the word containining the list of fast event channels assigned to the process, `pds$special_channels` with the event mask for the current validation level. That event mask is held in the array, indexed by ring, `pds$event_masks`. Technically, it would be sufficient to simply return the value of `pds$event_masks` for the current validation level, but in case of any residual values in the event mask, or inconsistency in the two values, computing the `AND` is proposed.

In keeping with standard Multics calling conventions for subroutines, the final argument is a status code, although in the above, it can be seen that this always returns 0 (denoting success) for the status code.

The third change is to declare the `hcs_$get_assigned_channels` entrypoint in `pl1.dcl`, where tools like emacs' `pl1dcl` command, and the `display_entry_point_dcl` (`depd`) command can look to find the definition of system subroutines.

The change is shown below:

```
Inserted in C:
C194     hcs_$get_assigned_channels entry (bit (36) aligned, fixed bin (35))
Nothing inserted in B
Preceding:
A194     hcs_$get_author            entry (char(*), char(*), fixed bin(1), char(*), fixed
```

```
    bin(35))

Comparison finished: 1 difference, 1 line.
```

The final change is to update the `hcs_.info` segment to include documentation for the new gate entry. That change is shown below.

```
cpa >doc>info>hcs_.info ==

A1       :Info: hcs_: 2022-08-26 hcs_
Changed by B to:
B1       :Info: hcs_: 2025-02-10 hcs_


Inserted in B:
B1184    :Entry: get_assigned_channels: 2025-02-10 hcs_$get_assigned_channels
B1185
B1186
B1187    Function: returns the fast IPC event channels that have been assigned
B1188    to this process at the current validation level.
B1189
B1190
B1191    Syntax:
B1192    declare hcs_$get_assigned_channels entry (bit (36) aligned,
B1193        fixed bin (35));
B1194    call hcs_$get_assigned_channels(special_channels, code);
B1195
B1196
B1197    Arguments:
B1198    special_channels
B1199       is where this gate entry returns the fast IPC channels. (Output)
B1200    code
B1201       is a standard system status code. (Output)
B1202
B1203
Preceding:
A1184    :Entry: get_author: 1982-03-08 hcs_$get_author


A4481                                                  END HISTORY COMMENTS */
Changed by B to:
B4501      2) change(2025-02-10,Swenson), approve(2025-02-10,MCR10138b),
B4502         audit(2025-02-10,GDixon):
B4503         Add hcs_$get_assigned_channels gate entry.
B4504                                                  END HISTORY COMMENTS */

Comparison finished: 3 differences, 27 lines.
```

4 Analysis of Problem

For those interested in the details of the analysis of the problem, this section goes into depth as to the findings.

When the system was in the hung state, operator console input was not possible. It was observed, however, that accounting updates were occurring at 15-minute intervals, as should be the case. This suggests that event call channel processing was working fine (in other words, that the Initializer process had, for some reason, called `ipc_$block`, and had processed event call channel wakeups in that call). The operator console was inoperable (that is, hitting the `Escape` character on the operator console produced no prompt for a command) and it was clear from the operator console logs that the `admin` command had executed (we were in admin mode), but the `ame` command to exit admin mode had not been executed. Since there was reasonable assurance that the dps8m simulator's handling of operator input (the stream of `autoinput` statements in the dps8m script to boot or upgrade the system) was working just fine, this suggested that the Initializer process was not seeing the event wait channel wakeups sent from ring 0 when the operator console had input for the Initializer process.

A few other interesting things were seen. The Root Logical Volume (RLV) disks that were used in the test case had absentee jobs in the absentee queue that were ready to run on boot, and which did start running when the Answering Service (AS) was brought up. The telltale signs of absentee process logins were seen in the operator console output, and one or two of the absentee jobs also recorded a logout. However, at least one of the absentee jobs appeared not to have completed. More on this later.

Having no options to execute any operator commands, nor being able to login from any interactive terminals, the only option was to perform an "execute fault", take a dump, perform an emergency shutdown, reboot the system, and analyze the dump using `analyze_mulitics` (`azm`). Analysis of the dump revealed many interesting things:

- The Initializer process was executing an IPC event call handler, `as_request_server_$wakeup`.
- The ring-4 Initializer stack showed that the Initializer was in "admin mode".
- Absentee processes that had run to completion, and had effectively logged out, were still in Active Process Table (APT), and had not caused LOGOUT messages to appear in the Answering Service (AS) log (and emitted to the operator console). They were, however, in the "stopped" state (as far as the scheduler was concerned), which means that they had called `terminate_process_`, and subsequently, `hcs_$stop_process`.
- There was a pending AS Request in the `as_requests.ms` message segment. This request corresponded to Utility.SysDaemon's `sac deleted_old_pdds` command, from its `start_up.ec`.
- The ring-4 Initializer stack showed that admin mode had called `listen_`, which eventually called `ocd_$get_line` (via `iox_$get_line`) to read an operator command, but that `ocd_$get_line` had called `ipc_$block` waiting on operator input. It was `ipc_$block` that called the `as_request_server_$wakeup` event call handler.
- The ring-0 segment `oc_data`, containing the operator console state showed that the `ame` command had been read into the `read_io` buffer, indicating that the Initializer process had actually called called into ring 0 to read a line from the console, and that the simulated IOM had issued a `terminate` interrupt that been handled by `ocdcm_$interrupt_handler`. That handler is responsible for causing an event wakeup to be sent to the Initializer process.

- The operator console wakeup was found to be in the ITT and threaded into the Initializer's process by way of `pds$itt_head`. This indicated that it had not been moved to the Initializer ECT's `ITT_MESSAGE` list.
- The Initializer process' Active Process Table Entry (APTE) had the `wakeup_waiting` flag set.

Further analysis revealed the following had occurred:

- The answering service had been fully brought up (we processed the `startup` Initializer ring-4 command).
- The dps8 script used to upgrade the system included the following sequence of commands:
  - `startup`
  - `admin`
  - `pause 120`
  - `ame`
  - `logout * *`
  - `admin`
  - `pause 60`
  - `ame`
  - `shut`
  - `die`
  - `y`
- Of these, the `startup`, `admin`, and `pause 120` commands had executed. The `ame` command, although provided to Multics by dps8, had not yet been executed.
- The Initializer process was in "admin mode", and had executed the `pause 120` command, and had invoked `iox_$get_line` on the operator console I/O switch to read the next command (`ame`).
- The operator console `ocd_$ocd_get_line` procedure, called `ipc_$block` to wait for a wakeup signaling that input was available (e.g. the `ame` command).
- The dps8 simulator had the `ame` command in its `autoinput` queue, and had signaled the appropriate `terminate` interrupt. The `ocdcm_$interrupt_handler` handler for that interrupt had caused a wakeup message to be queued in the ITT and had resulted in the Initializer process being flagged for a pending wakeup.
- The Initializer's process had the ITT event message in its ITT list (`pds$itt_head`).
- While in the `ipc_$block` call (part of user-ring IPC), and waiting for the operator console wakeup, IPC noticed that there was at least one wakeup for a registered event call channel. That event call channel was associated with the handler `as_request_server_$wakeup`. This is the handler for so-called *AS Requests* – requests from user or daemon processes for something to be done for them by the Initializer process.

- ipc_$block (actually ipc_real_$full_block) called as_request_server_$wakeup, which is why we saw this procedure on the stack, to handle the AS request.
- as_request_server_$wakeup had checked the as_requests.ms message segment for the AS request to handle and noticed it was one of the requests that must be deferred if the Initializer process is in admin mode – which it was. The request was the sac delete_old_pdds command in Utility.SysDaemon's start_up.ec.
- When deferring an AS request, as_request_server_$wakeup leaves the request in the message segment, and sends a wakeup to the Initializer process (itself) to remember to process this request later (once out of admin mode).
- as_request_server_$wakeup then returns to its caller (ipc_real_$full_block).
- ipc_real_$full_block looks for more event call channel wakeups and finds the one just sent by as_request_server_$wakeup, and handles it by calling that same procedure again.
- as_request_server_$wakeup again defers the pending request, and sends itself a wakeup to process this later.
- We are now in a tight loop. We are repeatedly handling an event call channel which causes another wakeup, which requires handling the wakeup.

The ring-4 stack trace looks like this:

```
azm: sk 234|0

Reverse trace of >pdd>!zzzzzzzbBBBBBB>stack_4 (Seg 234)
Number of stack frames 18.
Stack begin = 234|2000 Stack end = 234|16100
FRAME      RETURN_PTR
234|14040  733|47261   >t>bound_as_misc_$as_request_server_|1343
234|12700  251|4232    >sl1>bound_ipc_$ipc_real_|3646
234|12520  251|335     >sl1>bound_ipc_$ipc_fast_|263
234|11520  256|2255    >sl1>bound_oc_$ocd_|2255
234|11040  263|6422    >sl1>bound_system_control_$sc_signal_io_handler_|272
234|10740  263|7611    >sl1>bound_system_control_$sc_admin_mode_|553
234|10400  313|16326   >sl1>bound_library_1_$signal_|1470
234|7620   726|4510    >sss>bound_misc_io_modules_$signal_io_|2414
234|7320   1077|34437  >sss>bound_command_loop_$listen_|471
234|6740   263|7746    >sl1>bound_system_control_$sc_admin_mode_|710
234|6420   722|7337    >sss>bound_ssu_$ssu_execute_|551
234|6160   263|11672   >sl1>bound_system_control_$sc_abort_line_util_|276
234|5220   727|2140    >sl1>bound_multics_bce_$command_processor_|2140
234|4200   722|4526    >sss>bound_ssu_$ssu_request_processor_|1010
234|3700   263|4457    >sl1>bound_system_control_$sc_execute_command_line_|543
234|2700   722|4134    >sss>bound_ssu_$ssu_request_processor_|416
234|2400   263|1637    >sl1>bound_system_control_$sc_process_command_line_|1175
234|2000   263|433     >sl1>bound_system_control_$system_control_|433
Previous stack frame 77777|1
```

Repeated runs of the test case, interrupted by performing an Execute Fault (XF) when the Initializer process appeared hung and a subsequent analysis of the dump, revealed that the Initializer process

was always executing as_request_server_$wakeup, which appeared at the top of the ring-4 stack, but that sometimes the process was executing message segment code in ring 1 or file system code in ring 0. This indicated that the Initializer process was not really hung but progressing through the AS request handler code. Adding some debug hphcs_$syserr calls to as_request_server_$wakeup showed that this event call handler was being invoked repeatedly, each time returning to its caller, ipc_real_$full_block, before being invoked again.

From the above information, as well as further examination of stack traces, ECT data structures, and reading code in user-ring IPC, it became clear what was happening.

When the Initializer process called iox_$get_line to read the next command (in admin mode) from the operator console, it called ipc_$block to wait for an event wait channel wakeup indicating that input was available. The user-ring IPC procedure that implements ipc_$block is actually ipc_real_$full_block. This procedure, despite its name, doesn't necessarily perform a "full block", that is, it doesn't necessarily call hcs_$fblock, to get ring 0 to put the process in the blocked state so it is not scheduled until the wakeup. In fact, ipc_real_$full_block checks for the existing of wakeups on the event wait channels passed to ipc_$block, as well as wakeups for all event call channels that have been registered. Only if no unprocessed wakeups are found, does it call ring 0. Even then, when it does call ring 0, the process may not block if there are wakeups in the Interprocess Transmission Table (ITT) for the process and current ring. In that case, hcs_$fblock may copy pending wakeups in the ITT to the process' ECT (for the current ring) and simply return to the caller (ipc_real_$full_block).

Now, in the specific scenario that manifested itself in the "hang", admin mode invoked iox_$get_line to read an operator command, which called ipc_$block to wait for the event. ipc_real_$full_block, found a wakeup for an event call channel was already in the ECT. This was a wakeup for as_request_server_$wakeup, the event call procedure for processing AS requests. These are requests from user processes to the Initializer process to perform some action. The invoking process creates an AS request message, and adds it to the as_requests.ms message segment and then looks up a published event call channel name and process ID in the Who Table (>sc1>whotab). It then sends a wakeup to that process over the corresponding event channel to indicate that there is an AS request to process. The process listening on this event call channel is the Initializer process. The specific request in this scenario was the sac delete_old_pdds command in the start_up.ec of Utility.SysDaemon, a daemon that is launched by the AS at startup time.

The AS request mechanism supports several different kinds of requests. One of these is configured to require deferral if the AS is in admin mode, which, in the current scenario it was. That request type is the executing of "admin commands", such as those initiated with the send_admin_command (sac) command. Therefore, the request from Utility.SysDaemon would have to get deferred. So as_request_server_$wakeup found the admin command request in the the message segment, and left it there because it needed to get deferred. In order to make sure that it would get processed later (and not forgotten), as_request_server_$wakeup sent a wakeup to its own process over the AS request event channel.

In so doing, the event call channel wakeup that was initiated by Utility.SysDaemon, was followed by a stream (never ending, as we'll see) of wakeups from Initializer to itself. As long as the AS was in admin mode, this stream would continue.

Turning our attention back to ipc_$block (actually ipc_real_$full_block), it is important to understand the sequence of events that occurred. First, recall that ipc_$block is on the stack because the admin mode listener asked to read an input line from the operator console, and at

the precise time that it did this, no wakeup was (yet) available. Since ipc_real_$full_block did not have a wakeup for the event wait channel passed to it by the ocd_ operator console I/O module, it looked through the ECT to see if there were wakeups for any event call channels. It found the wakeup from Utility.SysDaemon for the AS request. It therefore called the event call channel procedure (as_request_server_$wakeup), which returned after sending another wakeup. ipc_real_$full_block then found this new wakeup (from itself), and called as_request_server_$wakeup again. This continued until the Execute Fault, which crashed the system to allow analysis.

Because the loop in ipc_real_$full_block never ran out of wakeups to process for event call channels, it never called into ring 0 to retrieve more events, and therefore never saw the wakeup from ring 0 indicating that the ame admin command was available. And therefore, it never returned to its caller, which would have read the ame command, and exited admin mode. As a result of staying in admin mode, the stream of wakeups for AS requests would never terminate, and pending operator console commands would never execute.

Recall that we had noticed that the absentee processes were in the stopped state, but that the AS had not done the normal processing it does when a process terminates. This is because terminate_process_ sends a wakeup to the AS over a per-process event call channel associated with the dialup_ event call handler. Because this wakeup is sent from a different process than the Initializer, ipc_real_$full_block would have to call into ring 0 in order to retrieve this wakeup. Because of the previously-described loop, this was never going to happen. Consequently, these absentee processes would remain in this zombie state.

Now, there was still a question in my mind, as I did this analysis, as to how the wakeups from Initializer to itself were seen by the loop in ipc_real_$full_block, since that code never called into ring 0 to read more events. To understand this, we diverge briefly to explain how event processing occurs.

In the normal case of one process sending a wakeup to another process, the wakeup is passed to hcs_$wakeup, which calls into ring 0. For a ring-0 device wakeup pxss is invoked to send the wakeup to the process. Ring-0 code then copies the wakeup into the ITT, and marks the target process as having a wakeup (so that it will get scheduled if not running). When that process next calls ipc_$block, it will perform the actions described earlier, and then call hcs_$fblock or hcs_$read_events. These gate entries into ring 0 will look in the ITT for any wakeups for the calling process, and if there are any, will copy them to the ECT for the process. There is an ECT for each ring, so the current ring of the process, when it calls hcs_$fblock, will dictate which ECT to copy events to. The ECT is located in the per-ring stack header for the calling process.

So in this normal case, a process won't see a wakeup until it calls hcs_$fblock (or hcs_$read_events). ipc_real_$full_block will call hcs_$fblock when it finds no events in the ECT.

However, there is an optimization in the case that a process receives a wakeup from itself. In this case, when hcs_$wakeup is called, ring-0 IPC notices the sending and receiving processes are the same, and simply copies the wakeup into the appropriate ECT's ITT_MESSAGE list. Thus, it is not necessary to call into ring 0 to notice these wakeups. It is only necessary that ipc_real_$full_block find the wakeup in the ECT and process it.

As a result of this optimization, the wakeups from Initializer to itself to handle the deferred AS request were immediately delivered to the ECT, and therefore immediately available for handling. However, the wakeup from ring 0 signalling arrival of operator input (the ame) command, was never

delivered to the ECT because the Initializer process never called `hcs_$fblock` or `hcs_$read_events`.

One more piece of information is relevant here. As mentioned earlier, `ipc_real_$full_block` calls `hcs_$fblock` when it has no wakeups to handle and no wakeup for the event wait list channels passed to it by its caller. `hcs_$fblock` is similar in function to `hcs_$read_events`, but the latter never puts the process in the blocked (waiting) state if no wakeups are available. It simply returns to its caller if there are no wakeups. This means that it is safe to introduce a call to `copy_itt_messages` (which calls `hcs_$read_events`) in the loop in `ipc_real_$full_block`. The proposed fix does exactly that. There is no risk of this call blocking the process – if there are no events found, then the call returns. The only overhead of `hcs_$read_events` in this case, is that of a gate call into ring 0 and its accompanying stack switch.

5 Documentation

The new gate entry, `hcs_$get_assigned_channels` is documented in the updated `hcs_.info` segment.

No documentation changes are needed for the changes to `ipc_real_` since those changes are internal implementation details of user-ring IPC.

6 Test

The changes will be tested by creating a new Multics System Tape (MST) with the updated code, and running it with the disk image that causes the symptoms described in this MCR. Further, this new hardcore will be run on GHM_Test for some time before being installed in the system and GHM booted with this hardcore.

The new `hcs_$get_assigned_channels` has been tested by writing a user program that invokes it, and verifying that the values returned by it are consistent with previous calls to `hcs_$assign_channel` and `delete_channel`. Furthermore, the new `ect.ec` exec_com, written by Gary Dixon and proposed in another MCR will also use this new gate entry and be an additional test of the entry.

7 Version History

2023-08-31	1.0	Initial version of the MCR.
2023-08-31	1.1	Fixed typos and integrated pre-review comments.
2023-09-01	1.2	Fixed typos and integrated pre-review comments.
2025-01-19	1.3	Additional fixes due to initial fix not completely addressing the issue.
2025-02-01	1.4	Updated to add `hcs_$get_assigned_channels` and `pl1.dcl`.
2025-02-10	1.5	Updated to add description of change to `hcs_.info`.