MCR10138
Fix bug in ipc_$block That Can Cause Process to Loop Indefinitely
Revision 1.2

Eric Swenson

September 1, 2023

1 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".

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. It had everything to do with the fact that the `admin` operator command was executed at the "wrong time", and just after the `startup` command was executed, 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 a bug in the user ring IPC mechanism, was never able to either execute that `sac` command, nor able to process the `ame` command to exit admin mode. 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 simply boot 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, invoked admin mode directly after bring 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, any wakeups for the event wait channels on which the process is blocked 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_$fblock`, the code in ring-0 copies the wakeups for that process in the ITT to process 2's Event Channel Table (ECT) (for the current ring). So if the event call channel wakeup were sent to a different process than the sending process, then `ipc_$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 ticket describing the original problem as reported is located here: https://multics-trac.swenson.org/ticket/313.

2 Proposed Fix

The proposed fix is very simple and involves forcing the loop in `ipc_$block` to call into ring-0 every so often – after some number of loop iterations. Adding a loop counter at the appropriate place, and calling `hcs_$read_events` every N iterations will ensure that `ipc_$block` retrieves any wakeups for event channels NOT originating from within its own process. The operator console wakeups are ring-0 (device) wakeups that do not get placed directly into a process' ECT (because when they are detected by hardcore, the executing process may not be the intended recipient of the wakeup). They are placed in the ITT as mentioned previously, only to be picked up and moved to the ECT when the recipient process calls `hcs_$read_events` or `hcs_$fblock`.

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_$read_events`) to retrieve new wakeups. It will therefore retrieve any pending wakeup for the event wait channel that is in the event channel wait list passed to ipc_$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.

Deciding on the value of N, the number of iterations in the `ipc_$block` loop, is a necessary step in implementing the proposed fix. The value cannot be too large, or the delay before handling the event wait channel wakeups will be long. In the failing test case described above, this would mean a delay in seeing and executing the `ame` Initializer command and the exiting of admin mode. Using a value that is too small might incur more overhead and delay timely handling of event call channels, especially those sent by the same process. Event call channels are frequently used for periodic events, and leverage timers (see timer_manager_). Calls from `ipc_$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), that 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 `compare_ascii` output shows the proposed changes:

```
cpa [lpn ipc_real_.pl1] ==

Inserted in B:
B164     dcl      block_loop_count  fixed bin;
Preceding:
A164     dcl      block_val         fixed bin (3);


A750
Deleted by B, preceding:
B751                   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:
B759     /* Check the channels for a pending event to satisfy this block. Keep track of
    iterations through this loop and
B760        every 5 iterations, call into ring-0 to retrieve any additional events. This
    prevents a flood of event call
B761        channel wakeups from our own process keeping us from ever retrieving new event
    wait channel wakeups, which
B762        might allow us to exit this loop. */
B763
B764                  block_loop_count = 0;
B765                  do while (check_channels = YES);
B766                       block_loop_count = block_loop_count + 1;
B767                       if block_loop_count = 5 then do;
B768                            block_loop_count = 0;
B769                            call read_new_events();
B770                       end;


Inserted in B:
B931     read_new_events:
B932          procedure();
B933
B934          if block_val ^= cur_ring
B935               then call cu_$level_set (cur_ring);
B936
B937          call hcs_$read_events (ipc_data_$fast_channel_events, ("0"b));
B938
B939          if block_val ^= cur_ring
B940               then call cu_$level_set (block_val);
B941
B942     end read_new_events;
B943     %page;
Preceding:
A922     /* Pickup any new event messages which have arrived since we last blocked */

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:
B961               then call read_new_events();
```

```
Comparison finished: 5 differences, 39 lines.
```

As can be seen, from the code above, the number of changes is larger than might be expected because it is necessary to call `cu_$level_set` before and after calling `hcs_$read_events`. Since the code to set/reset the validation ring level, and invoke `hcs_$read_events` already existed in this program, the above change makes it into an internal procedure so it can be called here, as well as where it was called originally. There is already an `on` unit established to make sure that the validation level is reset upon unexpected conditions.

3 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 that IPC wakeups were not masked). 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_$read_line` (via `iox_$read_line`) to read an operator command, but that `ocd_$read_line` had called `ipc_$block` waiting on operator input. It was `ipc_$block` that called the `as_request_server_$wakeup` event call handler.

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_$read_line` on the operator console I/O switch to read the next command (`ame`).
- The operator console `read_line` procedure, called `ipc_$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 thus did have a command to read from the operator console.
- 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_$read_line` to read the next command (in admin mode) from the operator console, it called `ipc_$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_$block`, as well as wakeups for all event call channels that have been registered. Only if no unprocessed wakeups are found, does it call ring-0. Even then, when it does call ring-0, the process may not block if their are wakeups in the Interprocess Transmission Table (ITT) for the process and current ring. In that case, `hcs_$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_$read_line` to read an operator command, which called `ipc_$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 deferred 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, or of a ring-0 (device) wakeup being sent to a process, the wakeup is passed to hcs_$wakeup, which calls into ring-0. The code that supports this gate entry copies the wakeup into the ITT, and marks the target process as having a wakeup (so that it will get scheduled if not running). When that process next calls ipc_$block, it will perform the actions described earlier, and then call hcs_$fblock. This gate entry into ring-0 will look in the ITT for any wakeups for the calling process, and if there are any, will copy them to the ECT for the process. There is an ECT for each ring, so the current ring of the process, when it calls hcs_$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_$wakeup is called, ring-0 IPC notices the sending and receiving processes are the same, and simply copies the wakeup into the appropriate ECT. Thus, it is not necessary to call into ring-0 to notice these wakeups. It is only necessary that ipc_real_$full_block find the wakeup in the ECT.

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 `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.

4 Documentation

No documentation changes are needed since this is an internal implementation detail of user-ring IPC.

5 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.

6 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.