MCR10095
Fix Operator Console Hang on Multi-CPU Systems
Revision 1.1

Eric Swenson

October 23, 2021

1 Problem Description

On multi-cpu systems, the operator console sometimes fails to echo characters after requesting its attention. Depending on the environment, this can happen more or less frequently. It may not have happened much when Multics was running on real hardware, but when running on the DPS8/M simulator, especially on systems with starved real CPUs, it can happen a lot.

The Multics ticket for this problem is: https://multics-trac.swenson.org/ticket/147.

One requests the attention of the operator console by typing <ESC> under the DPS8/M simulator. On real hardware, the operator would have hit the REQUEST (on a 6001 or 6004 console) or RETURN key (on a 6601 console). The system responds with a prompt, which defaults to "M->". At this point, when everything is working normally, the operator can type a command, which is echoed by the console. The operator terminates the command with the REQUEST or RETURN key, whereupon it is executed, and any output printed on the console.

This MCR presents a solution to the intermittent problem, where the operator types <ESC>, the system prints the prompt, but then no typed characters are echoed, and any completed command is not executed, for up to 30 seconds. Normally, after 30 seconds, the system operates as expected – until the next time the hang occurs. On some systems, this problem happens quite frequently; I've seen it happen every few commands, and sometimes for a succession of commands.

2 The Reason for the Hang

At a high level, the reason this occurs is that one process holds the operator console data (`oc_data`) lock, and another process fields the "terminate" interrupt associated with the IO command that writes the prompt string to the console. Each IO (initiated with a CIOC instruction) is terminated when the IOM sends a processor the "terminate" interrupt. The interrupt handler (in `ocdcm_$interrupt_handler`) is called by `io_manager`, when the interrupt comes in. This handler attempts to lock the `oc_data` lock, and finds it locked by another process. In this case, it simply dismisses the interrupt (returns from the interrupt handler) without making note of the fact that the terminate interrupt came in.

The following sequence events is supposed to occur:

- The operator types <ESC>
- The IOM signals a "special" interrupt to the CPU.
- The special interrupt handler sends a wakeup to the process waiting on the event channel associated with the console. This will be the Initializer process which is using standard `iox_` entries to read and write to the console. The `iox_` DIM used is `ocd_`, which is a user-ring interface to the `ocdcm_` (ring-0) implementation of the operator console interface.
- The Initializer process, which has been blocked on input from the operator console, wakes up, and invokes an `iox_` entry to read a line from the console.
- `odc_` calls `ocdcm_` to "queue the io" and then process the IO.
- `ocdcm_` realizes that it needs to write a prompt and initiates an IO (again, a CIOC) with a command to write a string to the console.
- The string is written to the console
- The IOM sends a terminate interrupt to the CPU to signal completion of the IO.
- `ocdcm_$interrupt_handler` is invoked to process the interrupt and signal the completion of the IO and the need to process the next IO. This will be the "read line" initiated by Initializer.
- `ocdcm_` issues a CIOC to perform the read call
- A line is read from the console.
- The IOM sends a terminate interrupt to the CPU to signal completion of the IO.

The above description is a simplification of what actually happens, but captures the important details.

As mentioned above, calls to the interrupt handler (`ocdcm_$interrupt_handler`) result in an attempt to lock the `oc_data` lock, and upon failure (due to another process' holding the lock), ignores the terminate interrupt. Now, if there were not some other mechanism to handle this, the console would hang forever. Fortunately, there is. The scheduler, `pxss`, has a table of entries to invoke at regular intervals to poll for certain events. One of these entries is `ocdcm_$poll_for_timeout`. This entry processes any pending IO that has been queued, and in the case at hand, will process the "read line" IO. This will result in the echoing of any characters typed thus far, and the execution of the command, if any.

The `pxss` polling of the operator console happens at approximate 30 second intervals. So the most the operator will have to wait before their command is echoed is 30 seconds. Still, this is an annoying and disruptive delay – especially if it happens several times in a session.

The curious reader might wonder why another process might be holding the lock when the terminate interrupt comes in. There are at least two important cases. First, the aforementioned polling entrypoint is invoked by the scheduler and this runs in whatever process is running on one of the CPUs. So the lock may be held (for a very short time) by the polling entrypoint. Secondly, any process might initiate output to the console – for example RCP messages or message coordinator output. Any of the output sent to the console requires queueing IO, processing IO, and waiting for terminate interrupts. All queueing and processing of IO is done with the `oc_data` lock held.

3 Special Interrupts

The above description made brief mention of special interrupts. These occur (for the console), under various conditions, most importantly when the operator types <ESC> (or REQUEST/RETURN). For some reason, the authors of `ocdcm_` added an interesting check when the `oc_data` lock is unlocked. It checks to see whether a special interrupt was NOT processed due to being delivered when the interrupt handler could not get the lock. Just prior to any unlocking of the `oc_data` lock the code checks for this condition, and if it has been flagged, processes any pending IO. The result is that special interrupts aren't lost, as terminate interrupt are.

4 Proposed Solution

The proposed solution is to do what is done for special interrupts. In the interrupt handler, if we cannot get the lock, we set a flag that means "I got a terminate interrupt, but couldn't process it because I couldn't get the lock". In the code that unlocks `oc_data`, we check that flag, and if it is set, prior to unlocking, we repeatedly process IOs until there are no more, and until the flag is no longer set (we reset the flag at each iteration).

Here is a fragment of the interrupt handler before the proposed changes:

```
if stac (addr (oc_data.lock), pds$process_id) then do;              /* only field
    terminates if we get lock.*/
    call wire_and_mask;                                             /* so unlock can
        unwire...          */
    if interrupt_level = 7 then oc_entry.got_special_int = true;    /* operator pushed
        RE(TURN QUEST) ... */

    call process_io_status ();                                      /* process any
        associated I/O status... */

    call process_io ();                                             /* process any
        waiting output...    */

    call unlock_oc_data ();                                         /* release lock,
        unwire and unmask... */
end;
else if interrupt_level = 7 then do;
    oc_data.no_lock_flags.got_special_int = true;
    oc_data.meters.queued_special_int_count = oc_data.meters.queued_special_int_count
        +1;
end;

return;
```

The proposed change here is to add:

```
 else if interrupt_level = 3 then do;
    oc_data.no_lock_flags.got_terminate_int = true;
end;
```

just prior to the `return;`. This makes note of the fact that we got a terminate interrupt (a level 3 interrupt is a terminate interrupt in this context).

The unlock code looks like this now (with a long block comment removed for clarity):

```
    unlock_oc_data:
    proc ();

    do while (oc_data.no_lock_flags.got_special_int);                  /* special
        occurred while we held lock..*/
        oc_data.no_lock_flags.got_special_int = false;                 /* reset the
            flag...                */
        oc_entry_ptr = find_oc_entry ("");                             /* find the
            bootload console...     */
        if oc_entry_ptr ^= null then do;                               /* only do if
            console available...  */
            oc_entry.got_special_int = true;                           /* say it came
                from there...        */
            call process_io_status ();                                 /* process any
                pending status...    */
            call process_io ();                                        /* attempt to
                start the I/O...      */
        end;
        else if ^oc_data.mc_io_enabled then
            oc_data.in_service = false;
    end;

    if ^stacq (oc_data.lock, ""b, pds$process_id) then do;             /* clear lock if
        it is ours...      */
        if oc_data.lock ^= ""b then                                    /* not ours, if
            not free crash...   */
            call report_error (PANIC, "unlock_oc_data", "Lock not mine.");
    end;

    call unwire_and_unmask;                                            /* unwire stack,
        accept interrupts... */

    return;

end unlock_oc_data;
```

The proposed change here is to rewrite the code to check for both special and terminate interrupts:

```
   unlock_oc_data:
    proc ();

    dcl special_interrupt_received bit (1) automatic init (false);

    do while (oc_data.no_lock_flags.got_special_int |
        oc_data.no_lock_flags.got_terminate_int);
```

```
                    /* special or terminate interrupt occurred while we held lock */

        /* reset the appropriate flag */
        if oc_data.no_lock_flags.got_special_int then do;
             special_interrupt_received = true;
             oc_data.no_lock_flags.got_special_int = false;
        end;
        if oc_data.no_lock_flags.got_terminate_int then
             oc_data.no_lock_flags.got_terminate_int = false;


        oc_entry_ptr = find_oc_entry ("");                              /* find the
             bootload console...     */
        if oc_entry_ptr ^= null then do;                                /* only do if
             console available...  */

             if special_interrupt_received then do;
                  special_interrupt_received = false;
                  oc_entry.got_special_int = true;                      /* say it came
                       from there...        */
             end;

             call process_io_status ();                                 /* process any
                  pending status...    */
             call process_io ();                                        /* attempt to
                  start the I/O...      */
        end;
        else if ^oc_data.mc_io_enabled then
             oc_data.in_service = false;

    end;

        if ^stacq (oc_data.lock, ""b, pds$process_id) then do;              /* clear lock
             if it is ours...      */
        if oc_data.lock ^= ""b then                                     /* not ours, if
             not free crash...   */
             call report_error (PANIC, "unlock_oc_data", "Lock not mine.");
    end;

    call unwire_and_unmask;                                             /* unwire stack,
        accept interrupts... */

    return;

end unlock_oc_data;
```

The above change will process the IO status (from the terminate or special interrupt) and then process any other queued IO, if either a special or terminate interrupt was received while the current process was holding the lock.

The introduction of the automatic variable `special_interrupt_received` is necessary to remember that `oc_data.no_lock_flags.got_special_int` was set, because we need to mark the `oc_entry`

with the oc_entry.got_special_int flag so that process_io_status can perform the needed logic to handle special interrupts. The automatic variable is necessary because we cannot reset the no_lock_flags at the end of the loop because this would increase the chances of race conditions where the interrupt handler would set these flags between the time the above code is processing status or IOs.

The full compare_ascii output of the proposed changes follow. Note that there are two segments changed – ocdcm_.pl1 and oc_data.incl.pl1. The include file change adds the new bit to keep track of unhandled terminate interrupts:

```
mbuild: cmp

---------- ocdcm_.pl1

compare_ascii >ldd>hard>source>bound_wired_1.s.archive::ocdcm_.pl1 ==

Inserted in B:
B689          else if interrupt_level = 3 then do;
B690               oc_data.no_lock_flags.got_terminate_int = true;
B691          end;
Preceding:
A689


Inserted in B:
B2701         dcl special_interrupt_received bit (1) automatic init (false);
B2702
Preceding:
A2698              /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
     * * * * * * */


A2709
A2710         do while (oc_data.no_lock_flags.got_special_int);                  /*
    special occurred while we held lock..*/
A2711              oc_data.no_lock_flags.got_special_int = false;                /* reset
    the flag...                */
Changed by B to:
B2714         do while (oc_data.no_lock_flags.got_special_int |
    oc_data.no_lock_flags.got_terminate_int);
B2715                           /* special or terminate interrupt occurred while we held
    lock */
B2716
B2717              /* reset the appropriate flag */
B2718              if oc_data.no_lock_flags.got_special_int then do;
B2719                   special_interrupt_received = true;
B2720                   oc_data.no_lock_flags.got_special_int = false;
B2721              end;
B2722              if oc_data.no_lock_flags.got_terminate_int then
B2723                   oc_data.no_lock_flags.got_terminate_int = false;
B2724
B2725
```

```
A2714                  oc_entry.got_special_int = true;                          /* say it
    came from there...        */
Changed by B to:
B2728
B2729                  if special_interrupt_received then do;
B2730                       special_interrupt_received = false;
B2731                       oc_entry.got_special_int = true;                     /* say it
    came from there...        */
B2732                  end;
B2733

Inserted in B:
B2739
Preceding:
A2720          end;

Comparison finished: 5 differences, 28 lines.
r 10:28 0.990 23

cpa [lpn oc_data.incl.pl1] ==

A107                        3 pad_no_lock_flags bit    (35)     unaligned,
Changed by B to:
B107                        3 got_terminate_int bit    (1)      unaligned,        /* we
    could not process this terminate. */
B108                        3 pad_no_lock_flags bit    (34)     unaligned,

Comparison finished: 1 difference, 3 lines.
```

---

5 Unfortunately, This Doesn't Completely Fix the Problem

Testing has indicated that the above-described fix ameliorates the problem significantly. Prior to the fix, on a 2-cpu system, I could "provoke" the hang by simply typing a few commands (e.g. "who") in succession. In normal running of the system, I would get a hang very frequently. Removing all but one CPU made the problem go away. Adding back a second or additional CPUs would cause the problem to occur again – on my particular Multics system, every few commands.

With the above fix, and running for many hours, and typing hundreds of commands on the operator console, I only saw one hang. So why does it still happen?

There is a race condition that the astute reader will have already noticed. If we check for the terminate flag at unlock time, and process all pending IOs, and then find the flag no-longer-set, we exit the loop, and unlock `oc_data`. If a terminate interrupt comes in between finding the flag not set and the call to `stacq` to unlock the lock, we will have missed the setting of the flag. The number of instructions between the end of the above-referenced `while` loop and the call to the PL/1 operator `stacq` is small. However, when running under the DPS8/M simulator on a real host machine with

other processes running, it is quite possible that there will be latency in scheduling the simulator thread that services the Multics CPU. For example, I'm running my 2-cpu Multics system on a Linode VPS with 2 real CPUs. I'm also running another Multics system on that VPS, as well as two ITS systems, various web servers, and some Linux processes. Consequently, there is contention for the real host CPUs to run the 2 Multics CPU threads in the simulator that runs one of my Multics systems. So it is quite possible that between the time the terminate flag is checked, and the time that the process holding the `oc_data` lock actually unlocks the lock, the host thread running the Multics CPU might lose the processor. There is very little that can be done about this, other than not running anything else on the host machine besides the Multics simulator threads.

It is my feeling that we needn't worry about this race condition. I ran a 2-cpu Multics system on an 8-core Thinkpad laptop, and never experienced any hangs at all – even without the fix. In that case, there was no CPU starvation, suggesting that the problem described herein is exacerbated by CPU starvation.

Interestingly, an old Multician (sorry, I'm not saying that he is old, only that he was a Multician in the old days, as I was) has said that this operator console hang problem was observed "back in the day" on real hardware. When it did occur, operators simply waited for the `pxss` polling routine to "fix things". The fix suggested in this MCR would have made the problem much less frequent on real hardware as well.

6 Additional Steps Required for this MCR

Because there is a change to the include file, `oc_data.incl.pl1`, `peruse_crossref` was used to see what other sources reference this include file. They are:

```
pcref oc_data.incl.pl1

References to oc_data.incl.pl1: (10/11/89 1134.9 pst Wed)
    structure_library_4_.cds

References to oc_data.incl.pl1: (10/12/89 1917.7 pst Thu)
    bce_console_io.pl1, iom_switches.pl1, ocd_.pl1, ocdcm_.pl1,
    syserr_real.pl1
```

`ocdcm_.pl1` will be recompiled because of the changes above. The only other segment that needs recompiling is `structure_library_4_.cds` so that the new field will be displayed (by `azm` and `rzd`). Because the other referring programs do not reference the new bit and because the layout of the `oc_data` structure is not otherwise modified, it is not necessary to recompile them.

It may be that reviewers of this MCR may request that all referencing programs be recompiled. If that is the case, I will do that as part of this MCR installation.

7 Testing

A new hardcore with the above fix has been created and a couple people have run it. I have been running it on one 2-cpu Multics system with no issues, and as I said, with only a single hang in many, many command input attempts. Further testing will involve running this under various test suites and on various systems for some time before it is installed in the system for the next Multics

release, MR12.8. Testing will involve normal usage, as well as repeated execution of commands on the operator console.

8 Documentation

There are no documentation changes for this change.

9 Version History

2021-10-22	1.0	Initial version of the MCR.
2021-10-23	1.1	As per review, change fix code to consolidate checks.