MCR10148
Add Workaround for Overflow Fault in Scheduler When Hibernating Host
Revision 1.5

Eric Swenson

February 26, 2025

1 Important Note

This version of MCR10148 has been updated a lot from the previous (1.1) version. The reason for this is that the crash that led to ticket https://multics-trac.swenson.org/ticket/228, and consequently to the first version of this MCR, was not explained in that MCR. In fact, it is still not clear why the crash occurred and thus some of the analysis and the proposed fix from the earlier MCR was not correct.

This version attempts to focus on the original crash, its causes, and a fix that is more appropriate to prevent a similar crash from occurring again. The original MCR mentioned that the updated code would also prevent a related crash due to hibernating the host running the DPS8M simulator, which, itself was running Multics. The fix proposed in this version of the MCR also addressses that problem.

This version of the MCR includes analysis by Charles Anthony, Gary Dixon, and Eric Swenson. Gary Dixon contributed significantly to the current fix to pxss.

2 Problem Description

Multics ticket https://multics-trac.swenson.org/ticket/228 describes a crash of a Multics system running under the DPS8M simulator on a Virtual Private Server (VPS) host. The underlying issue, an Overflow Fault in the Multics scheduler, has also been seen on laptops running the simulator upon resuming the host operating system after a long period of hibernation.

The bug description resulted from a dump analysis of a system crash on Eric Swenson's Gold Hill Multics (GHM) system on March 1, 2021. Details of the dump analysis are presented later in this MCR. GHM was running in a 2-CPU Multics configuration with a near R2.0 version of the DPS8M simulator.

The direct cause of the crash was an overflow fault while incrementing the global "bank acount" of quantum credits by the computed amount of virtual CPU time used by the process that was running at the time the scheduler was invoked to schedule a new process on a given CPU. In the

crash scenario, this computed virtual CPU time was an extraordinarily (and abnormally) large value, resulting in the overflow when added to the bank account of quantum credits.

An identical crash has been seen after suspending a laptop with a running Multics. In addition, the issue can be artifically triggered by setting the host clock forward by many hours.

3 Analysis

Analysis of the issue at hand included an analysis of the crash dump of Multics after the overflow fault occurred.

```
azm 2.5 (abbrev)
> ERF 030121.0630.0.7 in directory >old_dumps dumped at 03/01/21 0630.3 pst Mon.
Proc  6 DBR 16625434 running      on cpu a  Backup.SysDaemon.z
```

The `analyze_multics` (`azm`) `why` request gives the primary reason for the crash, a fault while running in an interrupts-masked environment. The fault was an overflow fault in the Multics process scheduler program: `pxss.alm` at instruction offset 4311.

```
azm: why

Crash Message:
   sys_trouble: Fault while in masked environment.

Inconsistency found in Dump Associative Memories.
         Bootload cpu is a
Fault while in masked environment
Overflow Fault (33)
By: 44|4311 bound_tc_priv$pxss|4311
Ref: 112|451 tc_data|451
         (rev) prev_sp not valid 231|7720 for frame 72|1220

Reverse trace of prds (Seg 72)
Number of stack frames 1.
Stack begin = 72|1220 Stack end = 72|1520
FRAME      RETURN_PTR
72|1220    34|2705     bound_interceptors$iom_interrupt|143
Previous stack frame 231|7720
```

The `events` request tells us that machine conditions describing the overflow fault were stored in the `pds` segment of the process running on CPU A at the `signal_data` offset within the `pds`.

```
azm: events
Events from 03/01/21 6:28:54.616112
   Time  CPU Proc Event                      Circumstances

54.616112 a      Fault: DRL                  RTB Machine Conditions
  .098015 b    0 Fault: CON                  prds$sys_trouble_data
  .097967 a    6 Fault: CON                  prds$sys_trouble_data
  .097922 a    6 Fault: OFL                  pds$signal_data
```

The machine_conditions (mc) request of azm displays information about the saved register contents, instructions and runtime environment:

```
azm: mc -pds signal -long

Machine conditions from pds$signal_data:
Pointer Registers:
   PR0 (ap) - 16|17514  as_linkage|17514
   PR1 (ab) - 231|7720  >pdd>!BLbCjxZbBBBBBB>stack_1|7720
   PR2 (bp) - 112|4200  tc_data|4200
   PR3 (bb) - 112|0     tc_data|0
   PR4 (lp) - 17|2612   ws_linkage|2612
   PR5 (lb) - 44|343    bound_tc_priv$pxss|343
   PR6 (sp) - 72|1220   prds|1220
   PR7 (sb) - 72|0      prds|0
Processor Registers:
   X0 - 1 X1 - 0 X2 - 4200 X3 - 3123
   X4 - 2 X5 - 0 X6 - 3344 X7 - 3406
   A Register - 000000000000 Q Register - 451307070725 E Register - 0
   Timer Register - 402022035 Ring Alarm Register - 0
Overflow Fault (33)
SCU Data:
By: 44|4311 bound_tc_priv$pxss|4311
Ref: 112|451 tc_data|451
On: cpu a (#0)
Indicators: cary, ovfl, ^bar
APU Status: priv, xsf, sd-on, pt-on, fabs
Instructions:
 20414   3 00451 0561 00    asq      pr3|451
 20415   3 00451 0561 00    asq      pr3|451
Mem Controller Mask: 000230000043 000400000000
MC Fault Time: 21-03-01 06:28:54.97922 pst Mon (153615605450112002)
Setting Temporary pointers from 71|140.
```

Note the ovfl flag is set in the Indicators word. This indicates an Overflow Fault.

A compilation listing of the pxss.alm program shows information about the instructions running when the overflow occurred, and provides information about what pxss was trying to accomplish by those instructions.

```
                            3871 "        COMPUTE_VIRTUAL_CLOCKS -- procedure to figure
                                 out what type of
                            3872 "        idle time we have, and also to update the
                                 time used in the
                            3873 "        APT entry for the process just run.
                            3874 "        A meter lock protects per-system doubleword
                                 variables
                            3890 "
                            3891 " " " " " " " " " " " " " " " " " " " " " " " " " " " " " "
                                 " " " " " " " " "
                            3892
   004300                   3893 compute_virtual_clocks:
```

```
004300 4a  4 00152 3521 20 3894          eppbp    pds$apt_ptr,*     set bp to this
    process
004301 4a  4 00262 2351 20 3895          lda      pds$page_waits    copy page fault
    count into APT entry
004302 aa  2 00002 7551 00 3896          sta      bp|apte.page_faults
004303 aa  2 00046 2371 00 3897          ldaq     bp|apte.virtual_cpu_time Remember
    bp's vcpu
004304 aa  6 00134 7571 00 3898          staq     temp              Use temp for delta
    virtual time.
                           3899
                           3900          read_clock
004305 4a  4 00104 6331 20               rccl     sys_info$clock_,*
004306 aa  3 00034 7571 00 3901          staq     bb|last_time      save last time
    anyone run
004307 4a  4 00324 1771 20 3902          sbaq     prds$last_recorded_time delta t in
    microseconds
004310 aa  6 00154 7571 00 3903          staq     delta_t
004311 aa  3 00451 0561 00 3904          asq      bb|governing_credit_bank
```

The overflow fault occurred in the above `asq` instruction. This set of instructions is trying to perform a variety of tasks related to:

- computing how much virtual CPU time has been used by the current process;
- computing how many wall-clock microseconds have past since pxss was last run on this CPU.

The `governing_credit_bank` variable is a system-wide count of the microseconds that have elapsed since the scheduler was last called on this CPU. Since each online cpu is tracking microseconds used on its CPU, this credit bank variable contains microseconds of the elapsed time that the currently running process consumed prior to being interrupted to possibly reschedule another process on that CPU. For example, on a 2-CPU system, if the process running in CPU A consumed 1 second of VCPU time, and the process running on CPU B consumed 0.5 seconds of VCPU time, then 1.5 seconds of VCPU time will get added to the `governing_credit_bank` variable. This variable is a single-word value, interpreted as a `fixed bin (35)` number.

You may wonder why this variable does not overflow more often. Yes, it accumulates elapsed microseconds used by each CPU very quickly. But just as quickly, those credits are passed out (the Multics code uses the term *scattered among*) the work classes that provide various guaranteed levels of performance. Each project can select and pay for a desired level of service (rate at which it can obtain virtual CPU time). The scheduler first assigns (or scatters) the `governing_credit_bank` microseconds among the defined work classes. The `getwork` procedure of the scheduler then assigns these virtual CPU credits as a VCPU quantum to each process assigned to that work class which is ready to run. If the process is ready to run AND has been assigned a quantum of VCPU time, then it is eligible to be run on a CPU. The highest priority eligible process is then assigned to run on the current CPU.

Credits in the *credit bank* are all assigned to the various work class entries (WCTEs) leaving zero credits unassigned in the `governing_credit_bank` (or even negative credits if over-scattering was done). That scattering should ensure that overflows dont occur. So why did GHM get an overflow fault that caused this crash?

The `pxss` code following the overflow instruction is shown below.

```
                            3905
004312 aa  2 00016 0771 00 3906          adaq     bp|apte.time_used_clock update time
    used in APT
004313 aa  2 00016 7571 00 3907          staq     bp|apte.time_used_clock
004314 4a  4 00260 1771 20 3908          sbaq     pds$virtual_delta update the virtual
    CPU time
004315 aa  2 00046 7571 00 3909          staq     bp|apte.virtual_cpu_time
                            3910
004316 aa  6 00134 1771 00 3911          sbaq     temp              Subtract old vcpu
    from new
004317 aa  6 00134 7571 00 3912          staq     temp              Remember for idle
    meters.
                            3913
004320 aa  6 00154 2371 00 3914          ldaq     delta_t           meter last
    recorded time
004321 4a  4 00324 0771 20 3915          adaq     prds$last_recorded_time
004322 4a  4 00324 7571 20 3916          staq     prds$last_recorded_time
```

This `pxss` code updates per-process meters recording virtual CPU time used by the current process. It also updates the timestamp for this invocation (`prds$last_recorded_time`) in preparation for use when `compute_virtual_clocks` is next called.

Notice several things about the code at lines 3900-3904 (where the overflow occurred).

```
                            3900          read_clock
004305 4a  4 00104 6331 20                rccl     sys_info$clock_,*
004306 aa  3 00034 7571 00 3901           staq     bb|last_time      save last time
    anyone run
```

The `read_clock` macro is reading the system's calendar clock, which returns the number of microseconds that have elapsed since Jan 1, 1901 0000. GMT. This returns a `fixed bin(71)` variable whose rightmost 52 bits contain the number of microseconds in the current clock reading. The clock reading is immediately stored at the `bb|last_time` location. So we can display the value in the dump. The `bb` pointer register is the same as the `pr3` pointer register, which we can see in the machine conditions points to `tc_data|0`. From the instruction word for the `staq` instruction (line 3901), we see the offset of `last_time` is 34o (34 octal). So our current clock reading is shown in the two octal words below.

```
azm: d tc_data|34 2
Segno 112 tc_data|34

   34     0 000000153615 605450111766
```

The left word would be contents of the `A` register, or `C(A)`; the right word the contents of the `Q` register, or `C(Q)`. The contents of the combined `AQ` register, or `C(AQ)` holds the `fixed bin(71)` variable which is stored by the `staq` instruction.

This location can also be displayed as a timestamp using knowledge of the header structure at the beginning of the `tc_data` segment. This structure is named `tcm` (for traffic control meters).

```
azm: d tc_data -as tcm.last_time
  last_time = 3792061734097910 2021-03-01 06:28:54.097910 pst
```

From the decoded timestamp, we can see that the `read_clock` returned a reasonable time value (within 2 seconds of time of crash, and within 12 microseconds of the time recorded in the overflow fault machine conditions).

The `sbaq` instruction at line 3902:

```
    004307 4a  4 00324 1771 20 3902          sbaq     prds$last_recorded_time delta t in
        microseconds
```

subtracts from this time value a timestamp that records when `compute_virtual_clocks` last ran on this CPU. This is referenced in the per-processor segment `prds` at location `prds$last_recorded_time`.

Unfortunately, we cannot display this earlier value as a timestamp because the header at the beginning of the `prds` segment is not a structure known to the `azm display` (`d`) request.

```
azm: d prds -as prds$last_recorded_time
azm (display): Syntax error in structure reference prds$last_recorded_time.
```

The code references this `prds$last_recorded_time` address through a snapped link in the linkage section of `pxss`. We can tell `azm` to indirect through this same link to display the two words of the timestamp, but only if we determine how this link was relocated when `pxss` was bound into the `bound_tc_priv` bound segment. It is usually easier to look at the offset of this entry point in the listing of the `prds.cds` data segment.

```
azm: ..pr >ldd>listings>prds.list -match last_recorded_time
       81     2 last_recorded_time fixed bin (71),              /* used by traffic control
             */
text|332       last_recorded_time
```

Examining that location shows:

```
azm: d prds|332 2
Segno 72 prds|332

   332      0 000000153615 134141021041
```

Notice that the left words of the the previous double-word timestamp and this one are both the same. So we can subtract just the `prds$last_recorded_time`s right word from that of the `tc_data|last_time` value to get the `delta_t`. Doing this manually reports the same value found in the `Q` register of the machine conditions.

```
Processor Registers:
   X0 - 1 X1 - 0 X2 - 4200 X3 - 3123
   X4 - 2 X5 - 0 X6 - 3344 X7 - 3406
   A Register - 000000000000 Q Register - 451307070725 E Register - 0
```

So the subtract worked correctly. This large `delta_t` is the cause of the overflow. Converting that `delta_t` to decimal, we can subtract that number of microseconds from the current clock reading obtained above.

```
azm: ..clock iso_date_time 2021-03-01 06:28:54.097910 pst -39914861013 microseconds
2021-02-28 19:23:39 pst
```

The large `delta_t` represents a time period of more than 11 hours! And adding that amount to the suspected value of `governing_credit_bank` prior to the crash is what's causing the overflow fault.

Looking back at the code that resulted in the overflow fault:

```
                          3899
                          3900          read_clock
004305 4a  4 00104 6331 20              rccl     sys_info$clock_,*
004306 aa  3 00034 7571 00 3901          staq     bb|last_time      save last time
    anyone run
004307 4a  4 00324 1771 20 3902          sbaq     prds$last_recorded_time delta t in
    microseconds
004310 aa  6 00154 7571 00 3903          staq     delta_t
004311 aa  3 00451 0561 00 3904          asq      bb|governing_credit_bank
```

we note that while `delta_t` is a two-word (fixed bin (72) unsigned) value, which can handle the large 11 hours time delta, the instruction that caused the overflow is only using the right-half of `delta_t`.

```
004311 aa  3 00451 0561 00 3904          asq      bb|governing_credit_bank
```

The value in `C(Q)` is really a `fixed bin(36)` unsigned value (though PL/I does not support that precision of `fixed bin unsigned`).

The value of `CQ` is `451307070725`, in octal, and as we can see, its left-most bit is ON. When interpreted as a fixed bin(35) value, that value becomes a large-magnitude negative number. If added to a perhaps negative `governing_credit_bank` value (which is permitted to become negative by over-scattering microseconds of VCPU time to the work classes), the addition of a large negative number to another negative number could certainly cause a fixed-point overflow.

Clearly the subtraction of the two timestamps has yielded a difference which is larger than will fit in the Q register.

So the root cause of the GHM crash is a very old timestamp in the `prds$last_recorded_time` for CPU A, more than 11 hours before the current timestamp. Why has this timestamp not been updated by subsequent calls to `compute_virtual_clocks`?

This is still a mystery, for none of the developers who have analyzed the issue has been able to come up with a plausible reason for this. Some of the suggested possibilities are:

- The DPS8M simulator returned a bogus value from the `rccl` instruction.
- The host computer system returned a bogus value to the simulator when the simulator was

performing the `rccl` instruction.

- The DPS8M simulator thread running CPU A somehow *froze* for 11 hours.
- The host computer's CPU, which was running CPU A somehow froze for 11 hours.
- Some data corruption occurred for the value of `prds$last_recorded_time`

None of these possibilities is very likely. And so far, we are stumped.

Neither the original version of this MCR, nor this version, includes a plausible explanation of how the timestamp in `prds$last_recorded_time` for CPU A could be so old.

Looking at the problem a slightly different way is helpful, although it still doesn't explain the crash. The `compute_virtual_clocks` of the scheduler (`pxss`) is called by `getwork`, another internal procedure. `getwork` is called whenever the currently running process is preempted (either by a timer runout when its quantum has expired or when preempted via a connect signal sent from another CPU). Whether an *idle* process or a non-idle process is running on a CPU, it will eventually get interrupted (pre-empted), and call `getwork`. So `compute_virtual_clocks` should get run on every CPU, at approximately 1/8 second intervals (the timer register is loaded with this value, causing the running process to handle a `Timer Run-out` (`TRO`) fault, which invokes the scheduler. Adding some tracing logic to the simulator such that it emits a trace message every time the value of `prds_last_recorded` time is updated (by `compute_virtual_clocks` shows this:

```
>>> CPU D Store 00072:000332(2) 000000157251 554647035157
>>> CPU A Store 00072:000332(2) 000000157251 554647037072
>>> CPU D Store 00072:000332(2) 000000157251 554647210613
>>> CPU E Store 00072:000332(2) 000000157251 554647211464
>>> CPU A Store 00072:000332(2) 000000157251 554647212042
>>> CPU D Store 00072:000332(2) 000000157251 554647213627
```

While that fragment doesn't show every CPU in the 5-cpu test system that generated them, a longer fragment would show that each CPU is calling `compute_virtual_clocks`. The values shown above are good and reasonable timestamps.

Consequently, unless one of the CPUs stopped running, or was deadlocked, the value of `prds$last_recorded_time` should get updated for all CPUs very frequently. How one value on GHM was left stale for 11+ hours is a mystery.

Because we do not understand the root cause of the crash, we cannot currently provide a repair for that problem. The best that can be done is to add an avoidance of the overflow(s) in pxss to prevent these Multics crashes.

4 Proposed Fix

After several discussions between Gary Dixon and Eric Swenson in investigating the root cause, Gary Dixon and Eric both felt it best to work on an avoidance of both the spurious (unknown cause) overflow in `pxss` on the GHM system, and of similar overflows in `pxss` caused when a laptop host running the Multics simulator was suspended because of laptop hibernation, thereby not permitting `governing_credit_bank` to be updated for many hours, or even for several days at a time. Both of these situations result in the same overflow when updating the credit bank. And both circumstances can avoid crashing by applying something like the change suggested below.

One possible solution would be to make the governing_credit_bank variable be a two-word, fixed bin (71) signed value. Because VCPU credits are *scattered* to the various work classes, changing the size of governing_credit_bank would also require changing the size of the work-class credit holding values. And since the work class credits are then distributed to processes in the work class, the APTE variables for the process would also have to be extended. This, coupled with the fact that on a properly running Multics system, the VCPU time given to a process prior to its being rescheduled should be very short (and is only long in the anomalous case due to a jump forward in time by the system clock), it doesn't make sense to increase the size of governing_credit_bank. A better way would be to throw out obviously bad values of delta_t (such as those resulting from an anomalous jump forward of the clock) and replace the VCPU delta with a reasonable value.

Consequently, the most straightforward way to avoid the crash involves the following changes:

- When compute_virtual_clocks is invoked, initialize a temporary to 0 to indicate we haven't made any adjustments to delta_t (yet).
- Make a copy of the low word of delta_t in a new temporary delta_t2 so that we can check its value and update it without affecting delta_t.
- Determine whether delta_t2 is greater than 5 seconds. If it is, then adjust it to 5 seconds. Otherwise, leave it alone. The 5 second value is the same value that is used when first starting up a Multics session to initialize the work class credit bank.
- Keep track in a meter when we do adjust delta_t2. Also keep track in the above-mentioned stack temporary that we made an adjustment (so we can check this later on).
- Use the value of delta_t2 to update the governing_credit_bank. The limited value of delta_t2 will prevent overflows of the credit bank (which is scattered to work classes before scheduling completes).
- Use the stack temporary that keeps track of whether we adjusted delta_t2 to skip the adjustment to the current process' apte.virtual_cpu_time, since the value is not an accurate reflection of the process' virtual cpu time.

The first change is to the pxss_page_stack.incl.alm include file to define temporary variables used in the updated version of pxss.alm. These include the delta_t2 and delta_t2_adjusted stack temporaries. delta_t2 holds a copy of the low word of delta_t, which is adjusted by the changes in pxss.alm when it is found to contain negative values or values that are considered too large.

```
cpa [lpn pxss_page_stack.incl.alm] ==

Inserted in B:
B6       "  2) change(2025-02-22,Swenson), approve(2025-02-22,MCR10148),
B7       "     audit(2025-02-23,GDixon), install(2025-02-23,MR12.9-1012):
B8       "     Updated to add new stack temporaries for delta_t2 and delta_t2_adjusted.
B9       "     Used by pxss in making adjustments to delta_t and in skipping updating
B10      "     apte.virtual_cpu_time.
Preceding:
A6       "                                                  END HISTORY COMMENTS


A38              temp     pad(21)           " to grow compatibly
```

```
Changed by B to:
B43               temp     delta_t2          " adjusted value of low word of delta_t
B44               temp     delta_t2_adjusted " flag to indicate whether we adjusted delta_t
B45               temp     pad(19)           " to grow compatibly

Comparison finished: 2 differences, 9 lines.
```

The next two changes add a meter in which to record overflow events in the `governing_credit_bank` updates done by `pxss`. A new `delta_t_overflow_count` meter is added to hold these counts.

```
cpa [lpn tcm.incl.pl1] ==

Inserted in B:
B7         2) change(2025-02-19,Swenson), approve(2025-02-23,MCR10148),
B8            audit(2025-02-23,GDixon), install(2025-02-23,MR12.9-1012):
B9            Add delta_t2_adjust_count meter.
Preceding:
A7                                                    END HISTORY COMMENTS */


A250         2 pad5 (174) fixed bin (35),                     /* room for expansion
    compatibly             */
Changed by B to:
B253         2 delta_t2_adjust_count fixed bin(35),           /* count adjustments to
    delta_t2 to prevent overflows */
B254         2 pad5 (173) fixed bin (35),                     /* room for expansion
    compatibly             */

Comparison finished: 2 differences, 6 lines.
```

```
cpa [lpn tc_meters.incl.alm] ==

Inserted in B:
B18      "  2) change(2025-02-19,Swenson), approve(2025-02-23,MCR10148),
B19      "     audit(2025-02-23,GDixon), install(2025-02-23,MR12.9-1012):
B20      "     Add delta_t2_adjust_count meter.
Preceding:
A18      "                                                 END HISTORY COMMENTS


Inserted in B:
B239              equ      delta_t2_adjust_count,338
Preceding:
A236
A237     "522 octal

Comparison finished: 2 differences, 4 lines.
```

The next change is to the `pxss.alm` scheduler, as described above:

```
cpa [lpn pxss.alm] ==

Inserted in B:
B113     "  7) change(2025-02-22,Swenson), approve(2025-02-22,MCR10148),
B114     "     audit(2025-02-23,GDixon), install(2025-02-23,MR12.9-1012):
B115     "     Updated to not allow values of delta_t used in virtual CPU calculations
B116     "     and WC credits to exceed 5 seconds. In this case, delta_t is adjusted to
B117     "     5 seconds and such adjustments are metered.
Preceding:
A113     "                                                  END HISTORY COMMENTS


Inserted in B:
B3905             stz      delta_t2_adjusted say we haven't (yet) adjusted delta_t2
Preceding:
A3900             read_clock


Inserted in B:
B3910             stq      delta_t2          save copy of low word of delta_t for possible
    adjustment
B3911
B3912             ldq      =5242880          alt_delta_t2 = 5*1024*1024 (5 seconds in
    microseconds)
B3913             cmpq     delta_t2
B3914             trc      delta_t2_ok       transfer if alt_delta_t2 >= delta_t2 &
    delta_t2 is not negative
B3915
B3916    delta_t2_adjust:                    " If  delta_t2 > alt_delta_t2 | delta_t2 is
    negative
B3917             stq      delta_t2          replace delta_t2 with the alt_delta_t2
    limiting value
B3918             aos      bb|delta_t2_adjust_count " increment counter of times we've
    adjusted delta_t2
B3919             lda      =o400000,du       set delta_t2_adjusted to negative to
B3920             sta      delta_t2_adjusted     remember that we adjusted delta_t2
B3921
B3922    delta_t2_ok:
B3923             ldq      delta_t2          add possibly-adjusted delta_t2 to
    governing_credit_bank
Preceding:
A3904             asq      bb|governing_credit_bank


Inserted in B:
B3926             ldaq     delta_t           get back full delta_t
Preceding:
A3906             adaq     bp|apte.time_used_clock update time used in APT


Inserted in B:
B3929
```

```
B3930             szn      delta_t2_adjusted did we adjust delta_t2
B3931             tpl      virtual_cpu_known if no, then avoid the next three instructions
B3932
B3933             fld      0,dl              store zero vCPU time into C(AQ) to do zero
    updates to idle meters
B3934             staq     temp              temp holds vCPU time for this pxss invocation
B3935             tra      virtual_cpu_time_zeroed
B3936
B3937    virtual_cpu_known:
Preceding:
A3908             sbaq     pds$virtual_delta update the virtual CPU time


Inserted in B:
B3944    virtual_cpu_time_zeroed:
Preceding:
A3914             ldaq     delta_t           meter last recorded time


A3929             adq      delta_t+1
Changed by B to:
B3960             adq      delta_t2


A3967             tnz      *+2
A3968             asq      bb|credit_bank    But others do.
A3969
Changed by B to:
B3998             tnz      *+3
B3999             ldq      delta_t2
B4000             asq      bb|credit_bank    But others do.
B4001             ldaq     delta_t           load back full value


A3972             lcq      delta_t+1         Decrement credits.
Changed by B to:
B4004             lcq      delta_t2          Decrement credits.

Comparison finished: 9 differences, 42 lines.
```

The change to the `traffic_control_meters` (`tcm`) command to display the new metering count is below:

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

Inserted in B:
B10
B11
B12      /****^ HISTORY COMMENTS:
B13        1) change(2025-02-19,Swenson), approve(2025-02-18,MCR10148),
B14           audit(2025-02-23,GDixon), install(2025-02-23,MR12.9-1012):
```

```
B15          Add meter of adjustments to delta_t2.
B16                                                    END HISTORY COMMENTS */
B17
B18
B19
Preceding:
A10      /* format: style3 */


Inserted in B:
B251                       /* Meter adjustments to large or negative delta_t2 values */
B252                       call ioa_ ("^16a^8d", "delta_t2 adjusts",
    cur_tcm.delta_t2_adjust_count);
Preceding:
A241                   end;

Comparison finished: 2 differences, 12 lines.
```

Finally, the info segment for `traffic_control_meters` is updated to document the new meter:

```
cpa >doc>priv>tcm.info ==

A1       02/28/85 traffic_control_meters, tcm
Changed by B to:
B1       :Info: traffic_control_meters: tcm:
B2       2025-02-19 traffic_control_meters, tcm


Inserted in B:
B151     The following meter shows the number of times we adjusted the value of
B152     delta_t2 to its being either too large or negative. It is shown if the
B153     -counters (-ct) control argument is supplied.
B154
B155     delta_t2 adjusts
B156        is the count of adjustments of delta_t2.
B157
B158
Preceding:
A150     The following meters pertain to the eligible queue. They are printed


Inserted in B:
B179
B180
B181     :hcom:
B182     /****^ HISTORY COMMENTS:
B183       1) change(2025-02-23,Swenson), approve(2025-02-22,MCR10148),
B184          audit(2025-02-23,GDixon), install(2025-02-23,MR12.9-1012):
B185          Add new delta_t2 adjusts meter to meter number of adjustments made by pxss
B186          to delta_t.
B187                                                    END HISTORY COMMENTS */
B188
```

```
At end

Comparison finished: 3 differences, 21 lines.
```

5 Documentation

The only documentation that is updated is that of the `traffic_control_meters` (`tcm`) command. The other changes are purely internal.

6 Test

The changes will be tested by running a hardcore generated with this fix on GHM and GHM_Test, as well as my laptop. The laptop will be hibernated and reawakened after several hours.

Furthermore, the `traffic_control_meters` (`tcm`) command will be run to ensure that it generates correct output.

7 Version History

2025-02-13	1.0	Initial version of the MCR.
2025-02-13	1.1	Incorporated some review comments.
2025-02-18	1.2	Rewritten due to new analysis and different proposed fix.
2025-02-23	1.3	A more optimized fix to pxss and updated to skip vcpu update with adjusted `delta_t`.
2025-02-24	1.4	Fix description of `pxss_page_stack.incl.alm` changes.
2025-02-26	1.5	Fix some typos and add a bit more explanatory text for why the proposed fix was chosen over an