Subject: MCR-10054, Fix Scheduler for Procs Required

Author: Eric Swenson

Date: May 4, 2019

Problem Description

The scheduler (pxss.alm) has a bug whereby it can preempt the wrong process when a ready-to-run process has limited the set of processors it can run on. When a waiting process is notified that it can run again, the scheduler searches for a process to preempt. However, it doesn't take into account the fact that the processor on which a candidate process is running might not be in the set of processors to which the now-ready process is restricted. It chooses the first running process to preempt, and once it has preempted it, it considers its task done. When the scheduler attempts to find a process to schedule on the preempted process' processor, it cannot schedule the ready process on this processor, so another process is scheduled. If no such process can be found, the idle process for this processor is chosen. The now-ready process must wait until to be scheduled, while some process running on one of its limited set of CPUs exhausts its quantum.

The bug manifests itself when Multics is configured to run more than one CPU and when one or more processes have been limited to run on a subset of the available CPUs. The set_procs_required program allows such a restriction. Since we do not have access to running Multics hardware today, we have seen the issue only when running under the dps8m emulator. The failing scenario can be witnessed in either a single-threaded or multi-threaded emulator, whether either the Multics CPUs are running in separate threads on one host CPU or on multiple host CPUs. We are confident that the issue would manifest itself on real hardware, as well.

As an example of the failing scenario consider a process (call it Process 1) that has used set_procs_required to limit the processors it can run on to just CPU B. Assume there are 2 processors on the system: CPU A and CPU B. Further assume that Process 1 (running on CPU B) takes a page fault and is placed in the waiting state until the required page is brought into memory. Further assume that some other process (Process 2) handles the terminate interrupt indicating that the faulted page is now in memory and its PTW is updated to reflect this. Process 2 invokes the "notify" logic in pxss to notify Process 1 that it can now be scheduled to run again. Pxss changes the state of Process 1 to "ready" (from "waiting") and then searches for a process to preempt so that the now-ready process can run again. Pxss loops through the entries in the Active Process Table (APT) looking for a running process. The first such running process it finds, if any, it preempts. That is, it causes the process to stop executing as soon as it is safe to do so. When the process yields the CPU, pxss looks for an appropriate process to run on the processor. In our scenario, Process 1 is ready, and could run on a processor, if that processor is in the set of allowed processors — in our case, CPU B. If, however, a process running on CPU A was located and preempted, Process 1 cannot run on this processor. Consequently, either some other eligible process is chosen to run on CPU A, or the idle process is run on that CPU.

There are two major cases to consider, but both have the same bad consequences. The first is when the tuning parameter gp_at_notify is set to false—the default. In this case, pxss searches only idle processes to preempt. It always starts with the bootload processor — let's say it is CPU A. If CPU A's idle process is found running, it is preempted. Process 1 cannot run on CPU A, and thus another process is scheduled (which may be the idle process again). The process running on CPU B — let's assume it is the idle process, is left running even though it should have been the process to be preempted. It runs until its

quantum is exhausted and a TRO fault is raised. At this point in time, pxss can schedule Process 1. But its resumption was delayed unnecessarily, until the idle process got a TRO. If no running idle process is found, the attempt to preempt a process proceeds with the next idle process. When at the end of the idle process list, pxss returns to the caller of the "notify" logic, having failed to preempt an appropriate idle process.

When gp_at_notify is set to true, pxss searches all eligible processes starting with the idle processes. The same situation as with gp_at_notify equaling false occurs. The first process to be checked is the idle process on the bootload processor (CPU A in our scenario). If it is found running, it is preempted, and the idle process on CPU B (if running) runs until TRO when it should have been preempted. Once all idle processes are searched, the eligible, non-idle running processes are checked. The first such process found is preempted, regardless of which processor it is running on—and no more processes are checked. If the preempted process is running on a processor on which the ready Process 1 is not able to run, then whatever process is running on an appropriate processor runs until its quantum is exhausted.

With either setting of gp_at_notify, there is no guarantee that the preempted process (if any) is running on an appropriate processor. If it is not, then the ready process cannot be scheduled, and must continue to wait until a process on an appropriate processor exhausts its quantum.

While the issue was discovered due to a large number of page faults causing a process to run much slower than if set_procs_required had not been used, the issue is not limited to processes becoming ready after a page fault has been satisfied. Any process that transitions from the waiting to the ready state may be delayed in resuming execution due to this issue. Other reasons a process might be waiting including explicit blocking (e.g. ipc_$block) such as when waiting for a signal or event.

Actual Scenario Found

The scenario described above is not theoretical and is easily replicated. We found the issue during testing of the Multics emulator with multiple Multics CPUs configured. We ran a program—cpu_test— on one CPU and then another, using set_procs_required to limit the process to one CPU or the other. We noticed an asymmetry in the execution times of cpu_test and the CPU utilization of the underlying host CPUs. The situation appeared to be most severe when a program running in a process takes a lot of page faults. We noticed that certain tests in cpu_test would pause for long periods of times and tracked the pausing tests down to those that took lots of page faults. We then wrote a test program (page_fault_test) that forced lots of page faults, and timed each iteration of a loop, where each iteration required a page fault to be signaled and handled before proceeding.

We noticed that when CPU A was the bootload processor, using set_procs_required for page_fault_test, where the processor on which it executed was limited to CPU B, page_fault_test ran significantly slower, with page faults taking 2-3 times longer than when the program was limited to running on CPU A, or when no processor limit was used. One confusing discovery was that when a CPU intensive program was running on CPU A, page_fault_test appeared to run at full speed on CPU B. During these tests, gp_at_notify was set to false — the default. This means that pxss only preempts idle processes.

The explanation for why a CPU intensive program running on CPU A caused "normal" behavior is that since pxss is only preempting idle processes, the only running idle process it would most frequently find to preempt would be the one on CPU B — the "right" one to preempt. This would allow the page_fault_test program to run quickly after preemption and there would be no delays.

When no CPU intensive program was running on CPU A, and the system was otherwise idle, the running process on CPU A was most frequently the idle process. And because CPU A was the bootload processor,

and because the bootload process is checked first, it was the one preempted, and it was running on the "wrong" processor.

When the bootload processor was switched from CPU A to CPU B, the "bad" behavior would be seen when the page_fault_test was limited to running on CPU A. Again, this can be explained because pxss always checks the bootload processor's idle process before any other idle processes. If the first idle process is found running, it is preempted, regardless of whether or not the now-ready process can run on that process.

Proposed Changes

The fix is actually a simple one and requires only a handful of instructions to implement. Since the bug is caused by preempting processes that are running on processors that the now-ready process is not allowed to run on, the fix is to check each candidate process for preemption to make sure it is running on one of the processors allowed to be used by the now-ready process. Any candidate process that is not running on an appropriate CPU is skipped. This means that no inappropriate process will be preempted (a gain in system performance) and only appropriate processes will be preempted. If there are no appropriate processes to preempt, then no process is preempted.

There are two parts to the fix. The first part is to remember the set of processors that the now-ready process can run on. In the code where the process' state is changed from waiting to ready, we simply get the procs_required_mask from the process' APTE, and store it in a stack temporary.

The relevant code can be found at the label gp_init in pxss.alm:

```
gp_init:  lda       bb|apte.thread,2    pre-empting on behalf of X2
          sta       temp                save (unique) threads for compares

          lda       bb|idle_tail        start with idle processes
          aos       bb|gp_start_count   For getwork/get_processor window
"
"All set up.  Now for the main loop.  It will be executed over the
"Idle processes first.  If no Idle process is running then if the
"XR0 is > 1 (ie: came in at get_processor entry,
"not get_idle_processor) then the loop will be executed over the
"elig processes.
"
gp_loop:  ldaq      bb|apte.thread,al   go higher in queue
          cmpa      temp                is this process which was notified?
          tnz       gp_tv,ql*           no - branch on state
          tra       gp_return           yes - give up and return
```

This code is changed to:

```
gp_init:  lda       bb|apte.thread,2    pre-empting on behalf of X2
          sta       temp                save (unique) threads for compares

          lda       bb|apte.procs_required,2      get procs_required for process being pre-empted
          ana       apte.procs_required_mask,du   mask out all but required processors
          sta       temp_procs_required           and save in our stack

          lda       bb|idle_tail        start with idle processes
          aos       bb|gp_start_count   For getwork/get_processor window
"
"All set up.  Now for the main loop.  It will be executed over the
"Idle processes first.  If no Idle process is running then if the
"XR0 is > 1 (ie: came in at get_processor entry,
"not get_idle_processor) then the loop will be executed over the
"elig processes.
"
gp_loop:  ldaq      bb|apte.thread,al   go higher in queue
```

```
          cmpa      temp                is this process which was notified?
          tnz       gp_tv,ql*           no - branch on state
          tra       gp_return           yes - give up and return
```

The three new instructions are shown, above, in blue. A new stack variable is defined in the include file pxss_page_stack,incl.alm:

```
         temp      temp_procs_required " to store preempting process procs_required
        temp      pad(21)             " to grow compatibly
```

The pad field is changed (as above) from 22 to 21. This means that no additional stack space is required — there were already 22 unused words of stack defined. Now there are 21 unused words.

The second part of the fix is to skip running processes that are running on a process not included in the temp_procs_required set. The relevant code is here:

```
gp_found_running:
          lxl3      bb|apte.thread,au   extract addr from next's backptr
          tsx6      LOCK_x3             gp locks
          ldq       bb|apte.flags,3     grab fresh copy of flags
          canq      apte.pre_empted,du  have we pre-empted it before?
          tnz       gp_skip             yes, skip it

          lda       apte.pre_empted+apte.pre_empt_pending,du turn on pre_empted bit
          orsa      bb|apte.flags,3
          ldq       bb|apte.flags2,3    get processor tag
          anq       apte.pr_tag_mask,dl
          cmpq      prds$processor_tag  this cpu?
          tze       gp_this_cpu         dont cioc if so
          cioc      scs$cow_ptrs,ql*
          tsx6      UNLOCK_x3           gp unlocks
          tra       gp_return
```

The code is changed as seen below, with the new instructions highlighted in blue.

```
gp_found_running:
          lxl3      bb|apte.thread,au   extract addr from next's backptr
          tsx6      LOCK_x3             gp locks
          ldq       bb|apte.flags,3     grab fresh copy of flags
          canq      apte.pre_empted,du  have we pre-empted it before?
          tnz       gp_skip             yes, skip it

          ldq       bb|apte.flags2,3    get processor tag
          anq       apte.pr_tag_mask,dl
          orq       =o400000,du          convert processor tag to bit position
          qrl       0,ql
          anq       temp_procs_required is prempting process allowed to use this cpu?
          tze       gp_skip             no, skip

          lda       apte.pre_empted+apte.pre_empt_pending,du turn on pre_empted bit
          orsa      bb|apte.flags,3
          ldq       bb|apte.flags2,3    get processor tag
          anq       apte.pr_tag_mask,dl
          cmpq      prds$processor_tag  this cpu?
          tze       gp_this_cpu         dont cioc if so
          cioc      scs$cow_ptrs,ql*
          tsx6      UNLOCK_x3           gp unlocks
          tra       gp_return
```

The inserted instructions check the process being examined, which has already been found running, to see if it is running on one of the processors on which the now-ready process can run. If it is not, then we skip to the next process. Otherwise, we proceed with the original code.

With the fix in place, the page faulting process (cpu_test or page_fault_test) runs at full speed on either the bootload CPU or any other CPU, regardless of whether set_procs_required is used or not. And when the test is run with a CPU-bound bootload processor, system and page_fault_test runtime is identical to the behavior without the fix.

Impact of New Instructions

These six additional instructions will be executed for each running process found while searching processes to preempt. Regardless of the setting of gp_at_notify, the number of such processes will be at maximum equal to the number of configured CPUs. The first such process exits the loop.

I considered adding a few more instructions to make the common case, where set_procs_required is never used skip the 6 instructions described above. Unfortunately, this adds about the same number of instructions. That short-circuiting code needs to do two things: It needs to remember that the now-ready process is using the default set of processors (there is an APTE flag that indicates this). Then, before the 6 instructions described above, it needs to check the whether the default set of processors was in use by the now-ready process and skip the 6 instructions above.

The most economical way to add the short-circuiting code is to add another stack variable to save the value of the "default_procs_required" flag from the now-ready process' APTE. This might look like this:

```
lda       bb|apte.flags,3                    grab fresh copy of flags
cana      apte.default_procs_required,du     mask out all but needed flag
sta        using_default_procs               store flag in stack (0=limited procs)
```

This adds three instructions upon entry to gp_init. Then we need the check:

```
ldq     using_default_procs                  do we need to check procs
tnz     skip_procs_required_check            no, we can skip check
```

The optimization adds 3 more instructions for all preemptions, for a total of 6. It adds 2 instructions for running candidate process checking, rather than 6 instructions (without the optimization). It isn't clear that the optimization is worth it.

In comparison tests of traffic_control_meters and total_time_meters using a 12.6f pxss versus a version with the fix as described in this MCR, but without the "optimizations" above, no discernible difference in performance, nor time spent in traffic control was detected.

Comparison of Code Pre-fix versus Post-fix

Without the optimization, a comparison of pxss before and after the fix appears below:

```
cpa <MCR10050>pxss.alm ==

Inserted in B:
B2775
B2776               lda       bb|apte.procs_required,2      get procs_required for process being
pre-empted
B2777               ana       apte.procs_required_mask,du   make sure we only have processors
B2778               sta       temp_procs_required           and save in our stack
B2779
Preceding:
A2775               lda       bb|idle_tail        start with idle processes
```

```
Inserted in B:
B2809
B2810               ldq       bb|apte.flags2,3    get processor tag
B2811               anq       apte.pr_tag_mask,dl
B2812               orq       =o400000,du
B2813               qrl       0,ql
B2814               anq       temp_procs_required is prempting process allowed to use this cpu?
B2815               tze       gp_skip             no, skip
B2816
Preceding:
A2804               lda       apte.pre_empted+apte.pre_empt_pending,du turn on pre_empted bit

Comparison finished: 2 differences, 13 lines.
r 16:27 0.278 0 level 2
```

Here is compare_ascii output for the change to the include file, pxss_page_stack.incl.alm, changed in order to define the additional stack variable:

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

A32
A33                 temp      pad(22)             " to grow compatibly
Changed by B to:
B32                 temp      temp_procs_required " to store preempting process procs_required
B33                 temp      pad(21)             " to grow compatibly

Comparison finished: 1 difference, 4 lines.
r 14:27 0.751 23
```

Reproducing the Problem

If reviewers are interested in reproducing the issue described in this MCR, they can do so using the emulator on the master branch of the Git repository https://gitlab.com/dps8m/dps8m.git. The R1.0 tag represents the released 1.0 emulator, and the issue can be seen in this version if the emulator is built from source with the ROUND_ROBIN and WAM C macros are defined (in Makefile.local).

The issue was found in a later version of the emulator (rc10-rc12) and can be seen by building the emulator from the rc12 branch with the LOCKLESS=1 option.

With the R1.0 emulator, explicit cabling configuration must be done to enable multiple CPUs. In the rc12 emulator, this cabling is default. In both cases, the Multics config deck must be updated to enable at least one additional cpu.

For those who want to reproduce using the R1.0 emulator, Charles Anthony has build instructions, and an emulator script that performs the required cabling and config deck updating. Ask for these on the dps8m-developers mailing list if you are interested.

Impact on Deadline Mode

Deadline mode, enabled by the deadline_mode tuning parameter but off by default, is a scheduling mode where processes in non-realtime work classes are scheduled according to the deadlines and quanta specified by their work classes. This is as opposed to the default behavior, where non-realtime work classes are scheduled according to the percentages specified for their work classes. Investigation into whether the changes proposed in this MCR would impact deadline model scheduling showed that there

would no interaction — deadline mode and notify/preemption are orthogonal and the changes would not change the behavior of deadline mode. Deadline mode only affects quanta computations based on work class configuration.

Documentation

No documentation is required for the fix to pxss because no user-visible effects results.

Testing

In addition to testing the scenario that caused us to noticed the anomaly with set_procs_required, I've run fix the fix applied to two systems (GHM and GHM_Test) for about a week (so far) continuously. One of these systems is a single-cpu system and the other is a two-cpu system. I've had no issues on either of these systems. I will continue to run these and other tests with and without set_procs_required for some time before installing the fix in the system libraries.

I've recreated the hardcore listings, run cpu_test, used the "hunt" command from the root to activate/deactivate all the segments in the file system, done SysAdmin tasks such as running the crank and bills, as well as performed normal edit/compile/debug cycles with the fix applied. I've run into no issues.

Version History

Date	Revision	Author	Comment
2019-05-04	1.0	Eric Swenson	Initial revision of the MCR.
2019-05-11	1.1	Eric Swenson	Corrected typos found by Gary Dixon
2019-05-18	1.2	Eric Swenson	Added some more information about the emulator environment needed to reproduce the issue.
2019-05-19	1.3	Eric Swenson	Update to clarify intro section a bit and add comment about deadline mode impact.
2019-05-19	1.4	Eric Swenson	Fix additional typos found by Gary Dixon and make minor changes suggested by him.