Priority Inheritance for CFS Bandwidth Control

Xi Wang - proposer, secondary code author and presentation author
Ben Segall - primary code author
Josh Don - design contributor and code reviewer
Hao Luo, Namhyung Kim, Stephane Eranian - performance monitoring
Non-kernel contributors: Alan Neads, Carlos Villavieja, Dan Gibson, Dawid Łazarczyk, Jiaen Ren, Radek Burny, Shiyu Hu, Sree Kodakara, Yiyan Lin, Xiangling Kong

Google Inc

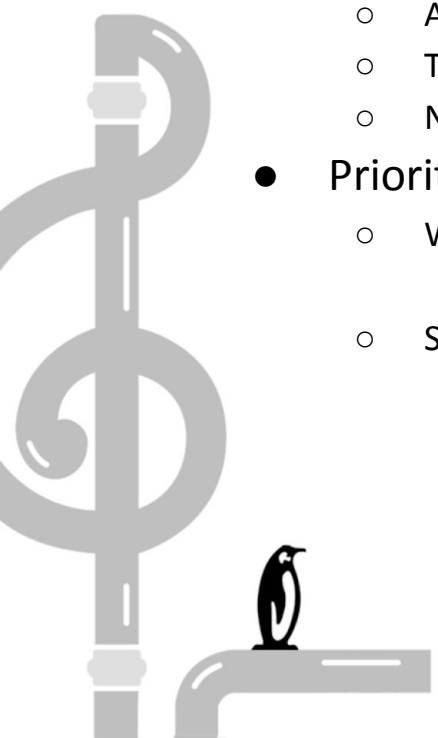
LINUX PLUMBERS CONFERENCE | Vienna, Austria
Sept. 18-20, 2024

Priority inversion without priority

- Deep throttling creates quasi priority among CFS tasks
  - Various methods and hacks for throttling
    - CFS bandwidth control with very low runtime/period
      - Our main server side throttling mechanism for various purposes
    - Tiny CPU cgroup shares
      - Throttle background tasks to save power on Android
        - **One of the motivations for proxy execution!**
    - Tiny cpu masks
      - Our previous memory bandwidth throttling mechanism
        - **Motivation for this talk!**
  - The quasi priority within CFS sched class
    - Normal - high priority
    - Throttled - low priority

Priority inversion without priority

- Same basic problem vs classical priority inversion
  - A throttled task holding a global kernel mutex, e.g. cgroup_mutex
  - The task's progress is slow due to throttling
  - Non throttled tasks has to wait for a long time
- Priority inversion is no longer limited to real time systems
  - We can get it as long as there is strong CPU scheduling differentiation
    - I think it is more generic than deadlocks.
  - Symptoms amplified by massive core count machines



Serious damaging effects

- Shared (multi-tenant) server, best effort jobs often needs to be throttled to make latency sensitive jobs perform better
  - Hardware membw throttling cannot throttle very low. CPU mask restriction was initially used
  - Thousands of threads could end up restricted to 2 cores
- Priority inversion was triggered
  - Throttled best effort tasks can be lining up taking a shared kernel lock and latency sensitive tasks have to wait
- Caused failures, not simply a performance problem
  - Wait time on shared locks can be really long due to high thread count
  - Machine managing agent timeouts - temporarily lost machines (lost contact or in watchdog triggered reboot) and end user visible application failures
    - Reducing best effort workload was the only effective workaround - lost machine capacity

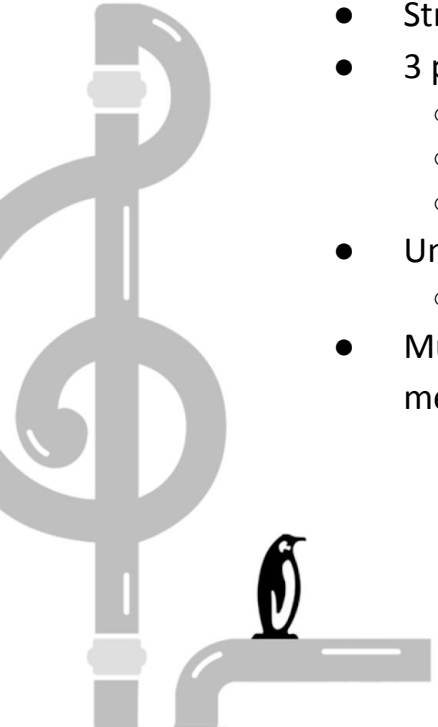
Real world vs textbook priority inversion

Textbook priority inversion

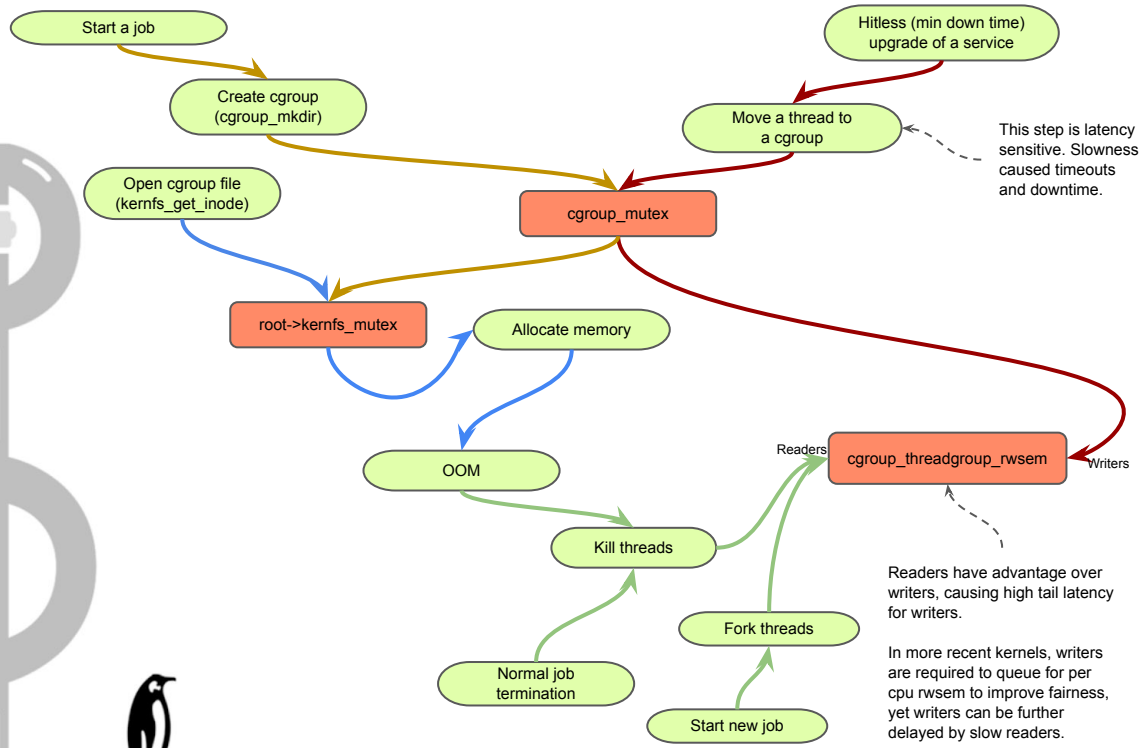
- Strict priority scheduling
- 3 priorities, 3 tasks
  - *High priority* waits on a mutex
  - *Low priority* holds mutex
  - *Low priority* preempted by *medium priority*
- Uniprocessor
  - 3 tasks on the same cpu
- Mutex holder is completely starved by the medium priority task

Production server priority inversion

- Non priority based CFS scheduling
- 2 quasi priorities, 1000s of tasks
  - *Regular* waits on mutex
  - *Throttled* holds mutex
- Multiprocessor
  - *Lots of tasks, regular* and *throttled* are usually on different cpus
- Mutex holder can still make slow progress, but the queue can be too long for mutex waiter



Sample lock dependency chart



Observations:

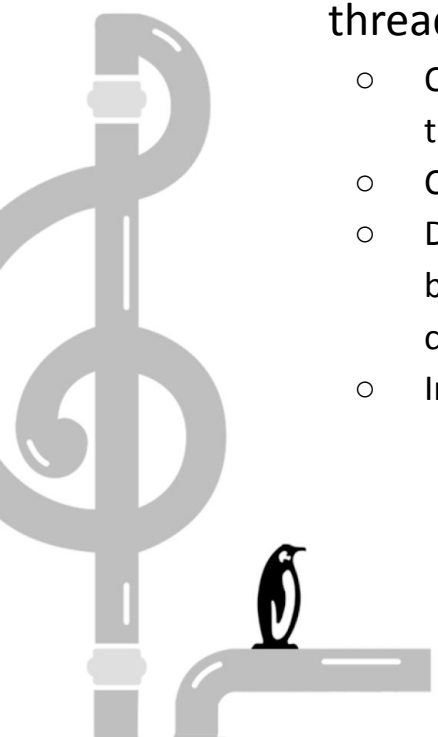
Writer latency is often more important than reader latency for cgroupfs: On a shared servers, writers are mostly machine managing agents starting/stopping jobs or make other adjustments. Readers can be stats collection or job specific actions. **Switching from mutex to rwsem can be counterproductive** (which pattern might be generic to control plane ops).

PI through both mutex and rwsem needs to be addressed. **If we leak a dependency path we can still get PI.**

In a non preemptive kernel, stuck mutex holders are often stuck at trying to get on cpu (runnable && !on_cpu) to actually accept a recently released mutex. **Out of order lock acquisition can be beneficial.**

Solution: Don’t throttle while in kernel mode

- Core idea: Treat **kernel mode as one big critical section**, don’t throttle a thread in kernel mode, throttling only happens in user mode
  - On a shared server, we mostly care about inter-job isolation. If we can prevent through-kernel priority inversion we should be mostly fine
  - Can also be considered a priority ceiling protocol
  - Don’t throttle in kernel -> lock holders making reasonable progress in kernel (and they can’t be holding kernel locks in userspace) -> solves priority inversion triggered by CFS bandwidth control
  - Implemented for CFS bandwidth control and deployed to our server fleet



Solution: Don’t throttle while in kernel mode

- Straightforward in concept, tricky in implementation, e.g.
  - A thread always wakes up in kernel mode, may need to unthrottle on wake up
  - How can we re-throttle when a thread returns to userspace
  - When a cgroup is allowed to run with negative cfs bandwidth control quota, only run kernel mode threads
    - Otherwise the user mode threads could get lots of temporary free quota and throttling fails
- Tracking whole kernel mode vs tracking specific locks
  - Tracking kernel mode means we can unthrottle threads that are not taking locks, but there should be no serious negative effect
    - CFS bandwidth control does account runtime debts / negative runtime so the free quota is only temporary - long term average of CPU bandwidth deliver is still precise
  - Tracking locks is more complex and some priority inversion dependencies don’t propagate through locks
    - E.g. anyone can write kernel code that checks a flag, set TASK_UNINTERRUPTABLE and call schedule()

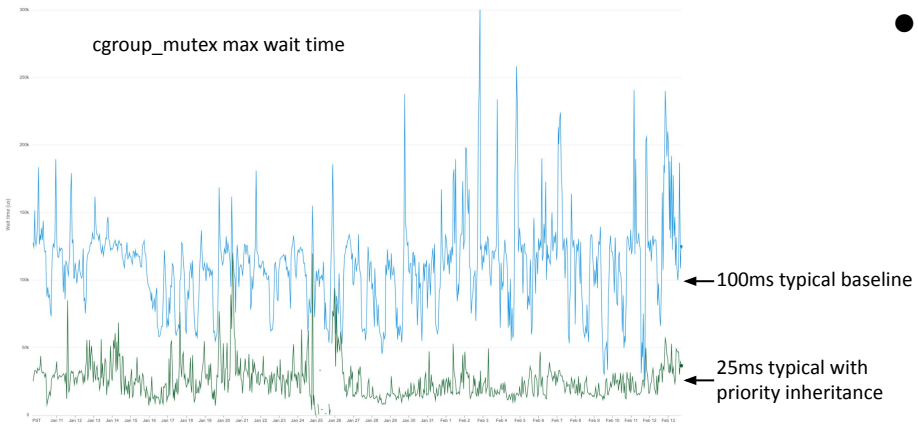
Parallel effort: "/when/ the throttling happens"

- Like some scientific discoveries (multiple discovery), there are actually two similar but independently conceived efforts. The other one is
  - Valentin Schneider discovered a deadlock scenario related to CFS bandwidth control while working on PREEMPT_RT
  - Peter Zijlstra then suggested
    - "I think 'people' have been pushing towards changing the bandwidth thing to only throttle on the return-to-user path."
  - Latest patch refer to the idea by "/when/ the throttling happens"
- We worked together
  - We started mid 2022 as a part time effort
  - Ben Segall saw Valentin's patch and posted our core patch
  - Valentin tried to merge our approach but ended up going back to his original approach due to cleaner implementation. Ben did some code review
- We can potentially switch to Valentin's patch
  - A major concern though is thread count scalability

Comparison based on my limited understanding

	CFS bandwidth control priority inheritance	"/when/ the throttling happens"
Motivation	Application timeouts	Kernel deadlock
Effects	Expected to be rather similar, both should solve cfsb related application timeouts and kernel deadlocks	
Domain of protection	Only lock holding critical sections or entire kernel mode. Entire kernel mode is currently used	Entire kernel mode
How throttling is temporarily avoided	Keep count for holding lock or in kernel mode status Don't start throttling with the count Unthrottle when a task with the count wakes up	Throttling is always deferred to exit_to_user_mode()
How re-throttling is enforced	On tick and at exit_to_user_mode()	Checked for each thread at exit_to_user_mode()
How to avoid unnecessarily giving cycles to user mode tasks	Maintain a kernel_list (second rq) of kernel mode tasks and only pick from it if a cgroup is runnable but should be throttled	Install task_work to every task if a cgroup should be throttled. curr gets throttled immediately if exit_to_user_mode(). Other runnable tasks get throttled if getting out of context_switch and exit_to_user_mode()
Scalability hotspot	Throttling / unthrottling task groups on presence / absence of kernel mode tasks, only when enabled	Enqueuing / dequeuing individual tasks on wakeup or exit_to_user_mode(), always present

Positive effects



- 2x~4x reduction of max lock wait time for kernel mutex and rwsems
  - Various application and machine managing agent timeout problems went away - tail cases were likely worse than the metrics suggested
  - Nearly eliminated timeout reboots from a one hour userspace watchdog
  - Collected with "perf lock con" feature from Namhyung Kim
- Translates to a machine capacity gain (or avoiding capacity loss)
  - More jobs can run on the same set of machines without breaking down

Further discussion: Magic all-generic mechanism?

	Specific to sched policy	**Oblivious to sched policy**
Specific to locking mechanism	Traditional priority inheritance	Proxy execution
Oblivious to locking mechanism	This talk and Valentin's patch	**Magic or a mess??**

Traditional priority inheritance: Strict priority scheduling, need to track lock owner - lock waiter relations

Proxy execution: Not specific to a sched policy, need to track lock owner - lock waiter relations

Work in this talk: Specific to CFS bandwidth control, no need to track locks

The 4th box: To be generic to both sched policy and locking mechanisms we can't differentiate at all and end up in a round robin scheduling mess?

Further discussion: Magic all-generic mechanism?

- Actually a mostly generic solution is possible
  - If we take it easy on performance requirements
- **Progress Guarantee Base Layer**: Guarantee a minimal rate of progress if in kernel mode
  - E.g. If kernel mode threads on a core add up to <10% of the core, run them to reach 10%
    - Can add a sched class independent pick_next mechanism but still need customized code for group scheduling and picking kernel mode tasks
    - Cycles can be delivered by a "DL Server"
  - Making sure lock owners always make progress limits the effects of priority inversion
  - Higher than CFS sched classes (SCHED_DEADLINE, SCHED_FIFO) already have bandwidth control mechanisms for preventing priority inversion. We can generalize by covering CFS based throttling mechanisms and having a unified mechanism for multiple scenarios
    - Higher cpu fraction for kernel mode and lower cpu fraction for user mode if desired
- Targeted optimizations can be added on top of the base layer

Further discussion: Complimentary lock optimizations

- Priority wait queues
  - Non FIFO queue for mutex waiters
    - A heavily contended lock often has multiple waiters in the wait queue, so why don't we give the lock to high priority waiters directly? This is more efficient than giving it to low priority waiters based on FIFO order then boost them
      - A shortcut compared to priority inheritance
    - Doesn't replace priority inheritance though as we can't take a mutex back from a low priority mutex owner
  - Straightforward for single mutex case but turns into a dependency tree if some tasks take multiple mutexes (similar to proxy execution)
  - More complex and less effective for rwsem
  - But a limited scope implementation might still improve performance

Further discussion: Complimentary lock optimizations

- rwsem is always more difficult due to 1:N dependencies
  - Writer tail latency is often more important for user-kernel interactions, e.g.
    - Periodically updating the stats - asynchronous and latency tolerant
    - Forking threads and putting them into cgroups - synchronous and latency sensitive
  - rwsem writers are more easily starved compared to mutex waiters
    - Not much we can do if a writer comes but 1 out of 100 readers is stuck
  - Recent mutex to rwsem conversions actually made our situation worse
- Idea: rwsem reader limit
  - If the estimated clear time of current readers is above a threshold, force newer readers to queue and wait
  - Writers have higher priority in wait queues
- Example
  - 100 concurrent readers on a rwsem. When the 101th reader comes, the estimated time of all 101 readers to clear the critical section is greater than 50ms so the 101th reader is queued
    - Estimate with past hold time histogram and/or a probability distribution model
  - When a writer comes, its wait time is likely less than 50ms