# A Case Study in Chaos Testing: Uncovering Kernel Scaling Issues

Bloomberg

Engineering

SREcon EMEA 2022
October 26, 2022

Gary Liku
Systems Reliability Engineer (SRE), Trading Systems Runtime

TechAtBloomberg.com

# Agenda

- Introduction
- Sporadic Scaling Problems
- Chaos Testing as an Investigative Tool
- The Problem is Threads... or is it?
  - Profiling /proc
- Mitigation Efforts

# Introduction

- Bloomberg Trading Systems Runtime SRE
  - Over 300 Managed Machines
    - Large hosts, 1TB RAM, 72 hardware threads
  - AIM & TOMS
    - Buy-side/sell-side order management systems
  - Shared Bloomberg Environment
    - Shared Memory
    - High volume of IPC
    - Farms for high CPU jobs
      - Suitable for vertical scaling
    - Precision Time Protocol (PTP) for certain time sensitive applications
  - Focus on a subset of high nproc Linux hosts

# Introduction

- New clusters added to our data centers
- Workloads migrated to take advantage of new hosts
- Things were going smoothly, until…

# Sporadic Scaling Problems

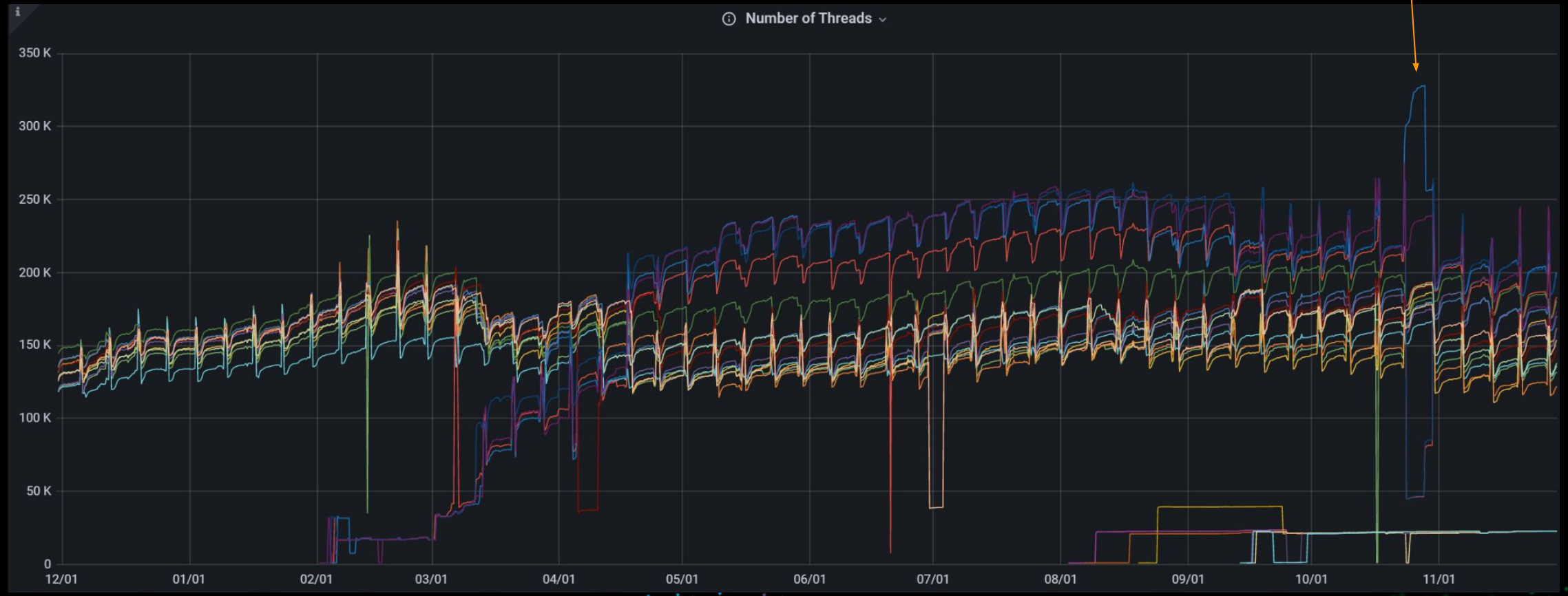
- Several seemingly unrelated issues became more common
  - Hosts getting hung or very slow
  - Services slow to start-up & shut-down during weekend load testing
    - Usually on scripts running ps or reading /proc
  - ps & /proc are used for monitoring, but they have a significant effect on the system
    - This is particularly bad!
  - Non-Interference Prime Directive for Visibility
    - *“The addition of any metrics acquisition subsystem should not noticeably affect the performance of the measured system”* (https://devops.com/of-max-and-min-the-non-interference-prime-directive-for-visibility/)

# Sporadic Scaling Problems

- More potentially correlated issues
  - Monitoring missing observations
    - More than ps provided metrics
  - PTP health check failing sporadically
    - Real-time requirements
      - Especially sensitive to load!
      - Needs 4 instances of clock drift within 1 minute to fail

# Sporadic Scaling Problems

What do our metrics tell us?



# Sporadic Scaling Problems

- Alerts correlated to high resource usage
  - Expected for heavy workloads
  - But might be an indication of resource contention

- Still unsure which resource that may be
  - How can we further investigate?

# Chaos Testing as an Investigative Tool

*“Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system’s capability to withstand turbulent conditions in production.”*

– principlesofchaos.org

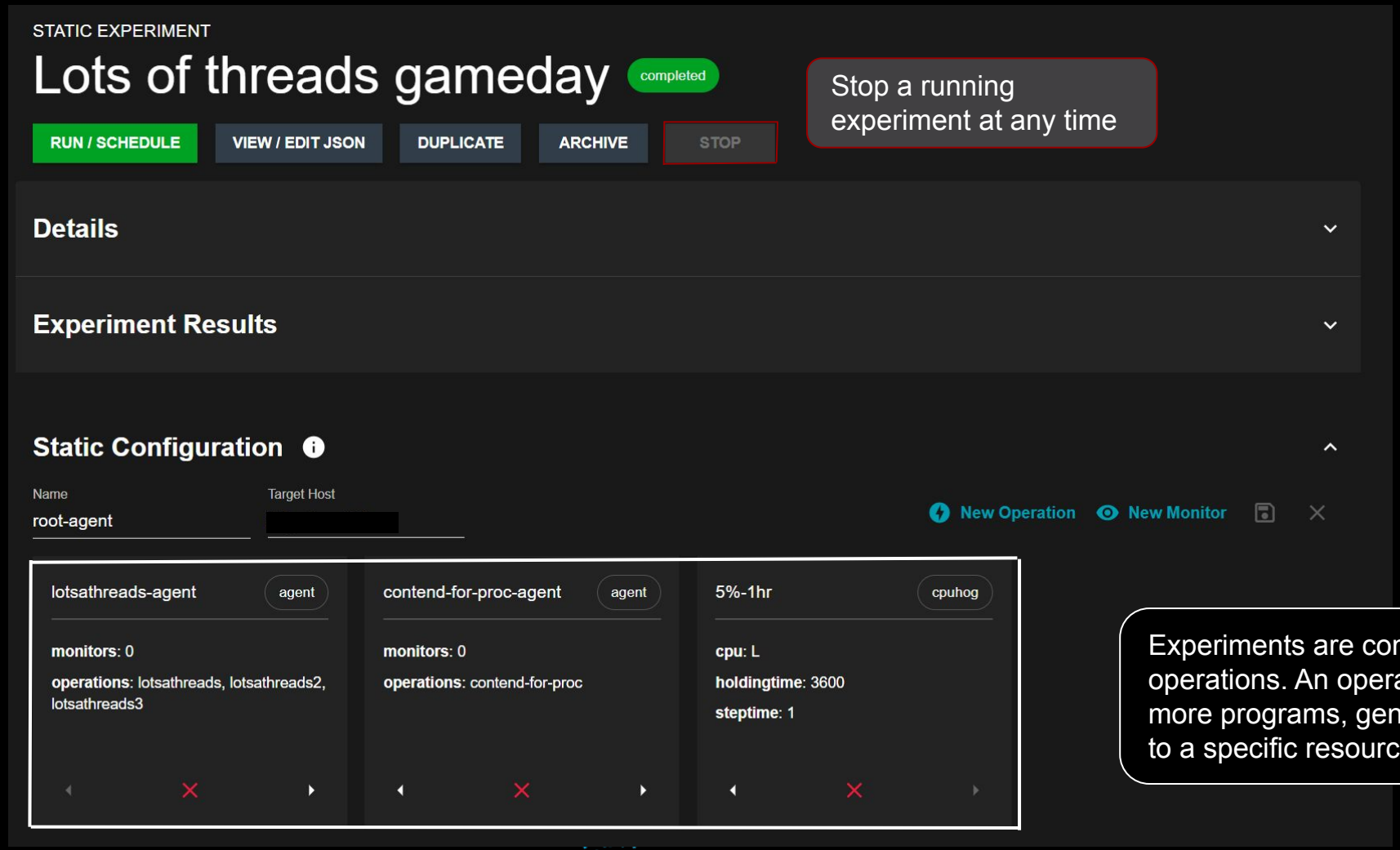
- But, we can also use it as an investigative tool
  - Signs of a resource contention issue
    - High load, high system CPU % utilization, high nproc
  - Use Chaos testing to replicate the contention
  - We can then isolate the root cause

# Orchestrating Chaos Tests

- How do we do it?
  - chaoscommander & generaldisarray - Bloomberg's chaos testing tools
    - chaoscommander for orchestration
    - generaldisarray as the executor on host
- One experiment per resource type (memory, CPU, nproc, etc.)
  - Simulate reading /proc load
  - Idle threads/procs for nproc test
- Reuse published metrics for test monitoring
  - Failsafe monitors based on metrics

# Preparing the Experiment

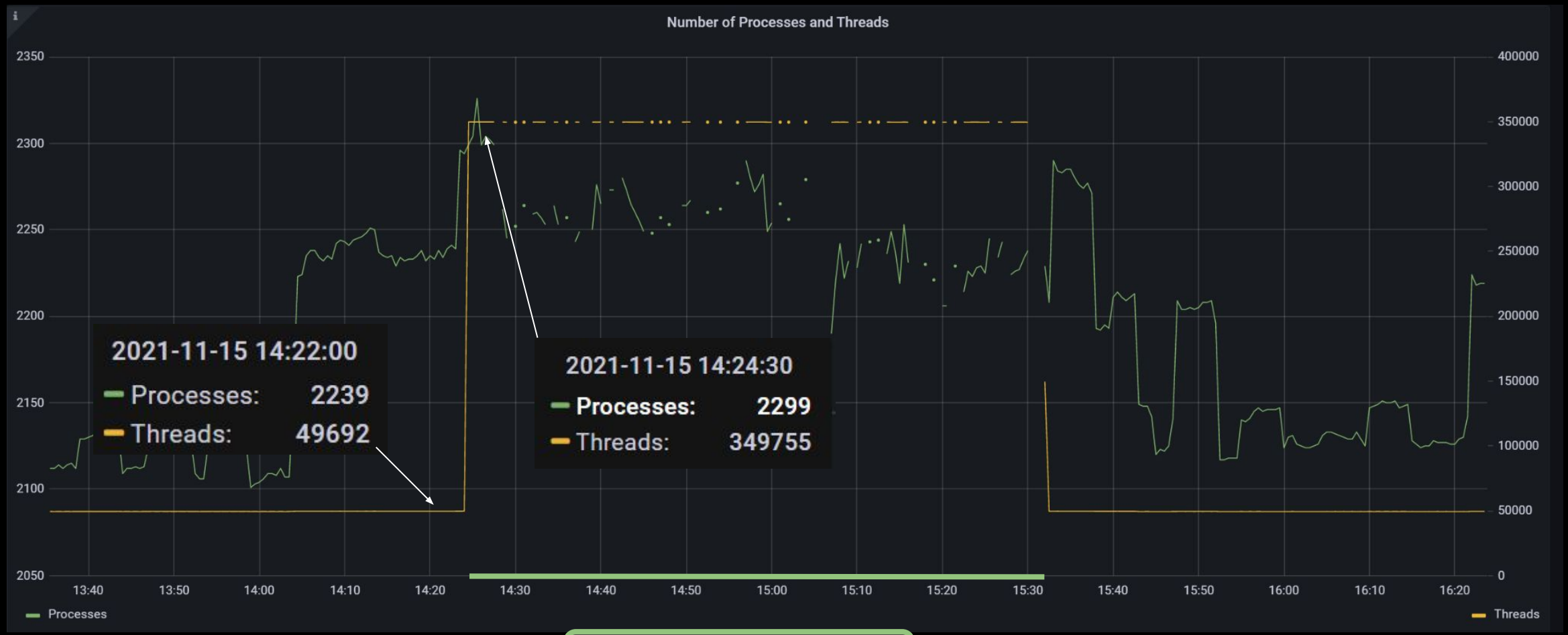
Run creates a ticket to start the experiment




Stop a running experiment at any time

Experiments are composed of operations. An operation is one or more programs, generally dedicated to a specific resource.

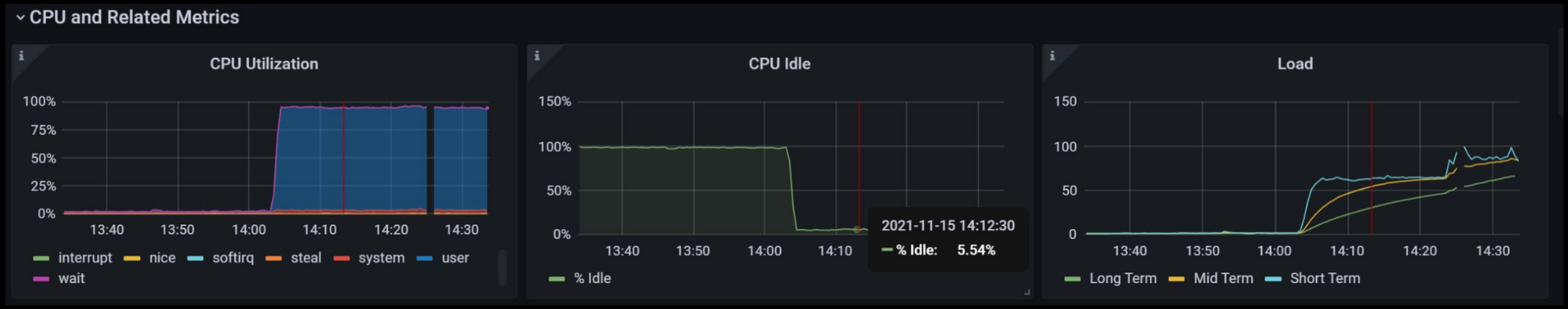
# Running the Experiment




Experiment Duration

# Running the Experiment

## CPUHog test



# Finishing the Experiment

- How do we know the experiment was successful?
  - Many of the  issues we were expecting emerged
    - 4 missed PTP checks
    - Gaps in monitoring
    - High system CPU with tasks hanging
  - On multiple hosts on the same resource test - threads
  - This was surprising, as we expect to scale close to ~4 million (pid_max)
    - But, we were seeing issues as low as 300k nproc
- Most importantly, repeat experiments gave the same results!

# Finishing the Experiment – Summary

- Observed several issues when scaling
  - PTP Alerts
  - Slow start/stop scripts - particularly those calling ps or reading /proc
  - Hanging/slowness
- Designed experiments to explore several potential causes
  - A test per resource - nproc, CPU, memory, etc.
- Experiments point to nproc as the culprit
  - Actually threads, as thread count >> process count
  - Alerts trigger ~300k nproc << pid_max (~4 million)
- Repeat experiments show similar results

# The Problem is Threads… or is it?

- /proc is an atypical directory; it's created on demand
- Linux kernel keeps track of processes & threads similarly
  - /proc/<pid>/task/<tid>

```
$ lotsathreads 10 &
[2] 1640548
$ ls /proc/1640548/task/
1640548  1640586  1640587  1640588  1640589  1640590  1640591  1640592  1640593  1640594  1640595
$ ps -L -o pid,ppid,tid,tgid,cmd -p 1640548
    PID    PPID     TID    TGID CMD
1640548 2423220 1640548 1640548 lotsathreads 10
1640548 2423220 1640586 1640548 lotsathreads 10
1640548 2423220 1640587 1640548 lotsathreads 10
1640548 2423220 1640588 1640548 lotsathreads 10
1640548 2423220 1640589 1640548 lotsathreads 10
1640548 2423220 1640590 1640548 lotsathreads 10
1640548 2423220 1640591 1640548 lotsathreads 10
1640548 2423220 1640592 1640548 lotsathreads 10
1640548 2423220 1640593 1640548 lotsathreads 10
1640548 2423220 1640594 1640548 lotsathreads 10
1640548 2423220 1640595 1640548 lotsathreads 10
```

# Profiling /proc

## proc_pid_readdir

- 3.10 Kernel
  - Ubuntu 13.X
  - Redhat 7.X

```
  /* for the /proc/ directory itself, after non-process stuff has been done
  */
  int proc_pid_readdir(struct file * filp, void * dirent, filldir_t filldir)
{
[..]
     for (iter = next_tgid(ns, iter);
          iter.task;
          iter.tgid += 1, iter = next_tgid(ns, iter)) {
          if (has_pid_permissions(ns, iter.task, 2))
               __filldir = filldir;
          else
               __filldir = fake_filldir;

          filp->f_pos = iter.tgid + TGID_OFFSET;
          if (proc_pid_fill_cache(filp, dirent, __filldir, iter) < 0) {
               put_task_struct(iter.task);
               goto out;
          }
     }
     filp->f_pos = PID_MAX_LIMIT + TGID_OFFSET;
out:
     return 0;
  }
```

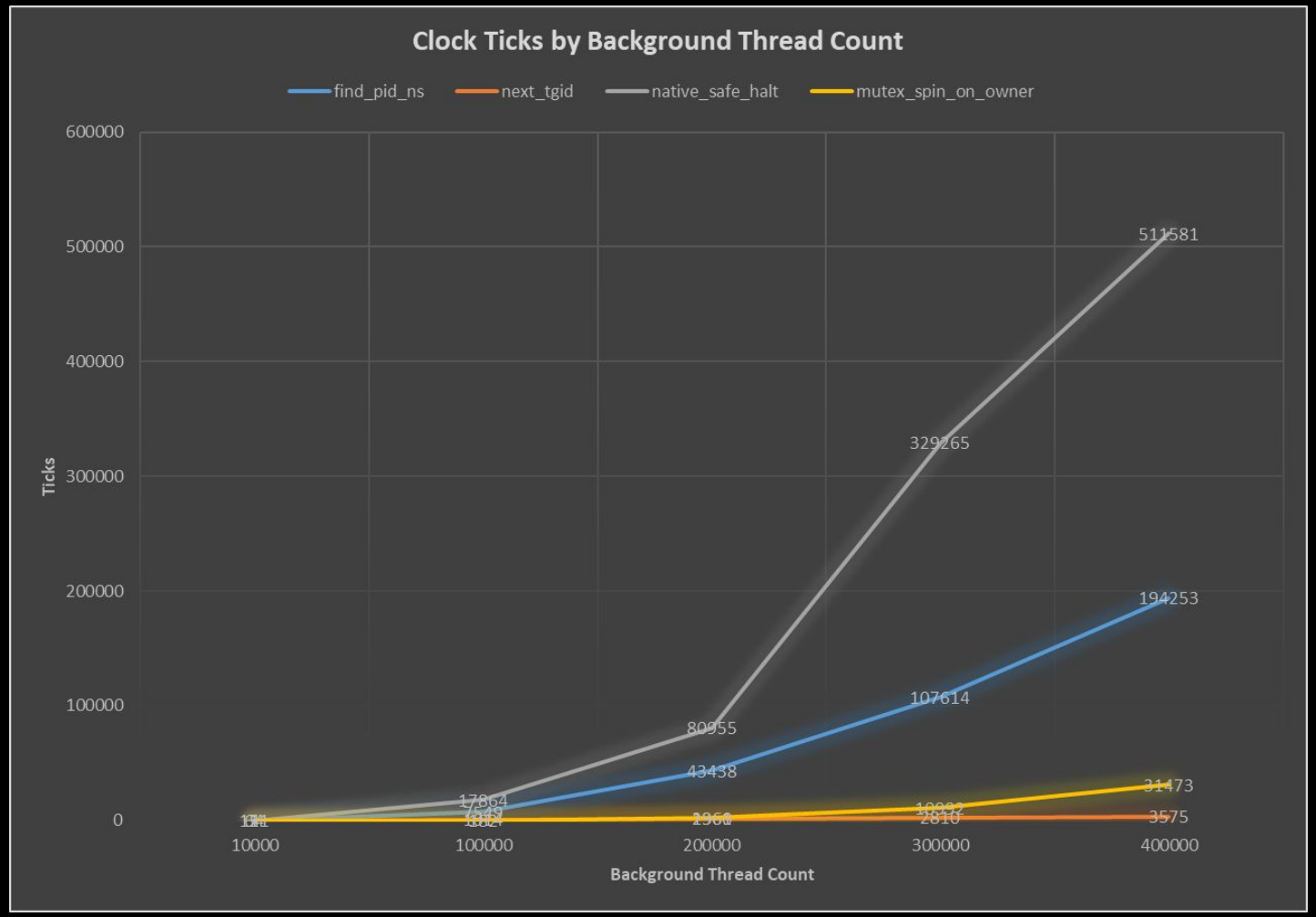
TechAtBloomberg.com



Engineering

# Profiling /proc

- Rerun the chaos test for threads with profiling
- Reset /proc/profile counters for each test
- Top 4 functions by clock tick
  - native_safe_halt
  - find_pid_ns
  - mutex_spin_on_owner
  - next_tgid




TechAtBloomberg.com



Bloomberg

Engineering

# Profiling /proc

## next_tgid called from proc_pid_readdir

```
static struct tgid_iter next_tgid(struct pid_namespace *ns, struct tgid_iter iter)
{
      struct pid *pid;

      if (iter.task)
            put_task_struct(iter.task);
      rcu_read_lock();
retry:
      iter.task = NULL;
      pid = find_ge_pid(iter.tgid, ns);
[..]
      rcu_read_unlock();
      return iter;
  }
```

```
/*
 * Used by proc to find the first pid that is greater than or equal to nr.
 *
 * If there is a pid at nr this function is exactly the same as find_pid_ns.
 */
struct pid *find_ge_pid(int nr, struct pid_namespace *ns)
{
     struct pid *pid;

     do {
          pid = find_pid_ns(nr, ns);
          if (pid)
               break;
          nr = next_pidmap(ns, nr);
     } while (nr > 0);

     return pid;
 }
```

next_tgid
-> find_ge_pid
-> find_pid_ns

# Profiling /proc

- Locks are a good candidate for the cause of contention
- We see contention and we see locks
  - The RCU locks cause contention!
- Problem solved? Not quite
  - RCU locks are designed to avoid blocking on readers, but allow blocking on writers
  - The data shows contention on readers (ls /proc, ps)
  - So, what's going on?

# Profiling /proc

"The problem is get_pid_list() traverses the entire tasklist in order to build the PID list needed by ls /proc. It read-holds tasklist_lock during this traversal and blocks updates to the tasklist, such as those performed by fork(). On machines with large numbers of tasks, this can cause severe difficulties, particularly given multiple instances of certain performance-monitoring tools."

– Paul E. McKenney (maintainer of Linux kernel RCU)
October 1, 2003
https://www.linuxjournal.com/article/6993

# Prime Directive Violation!

LAWFUL NEUTRAL

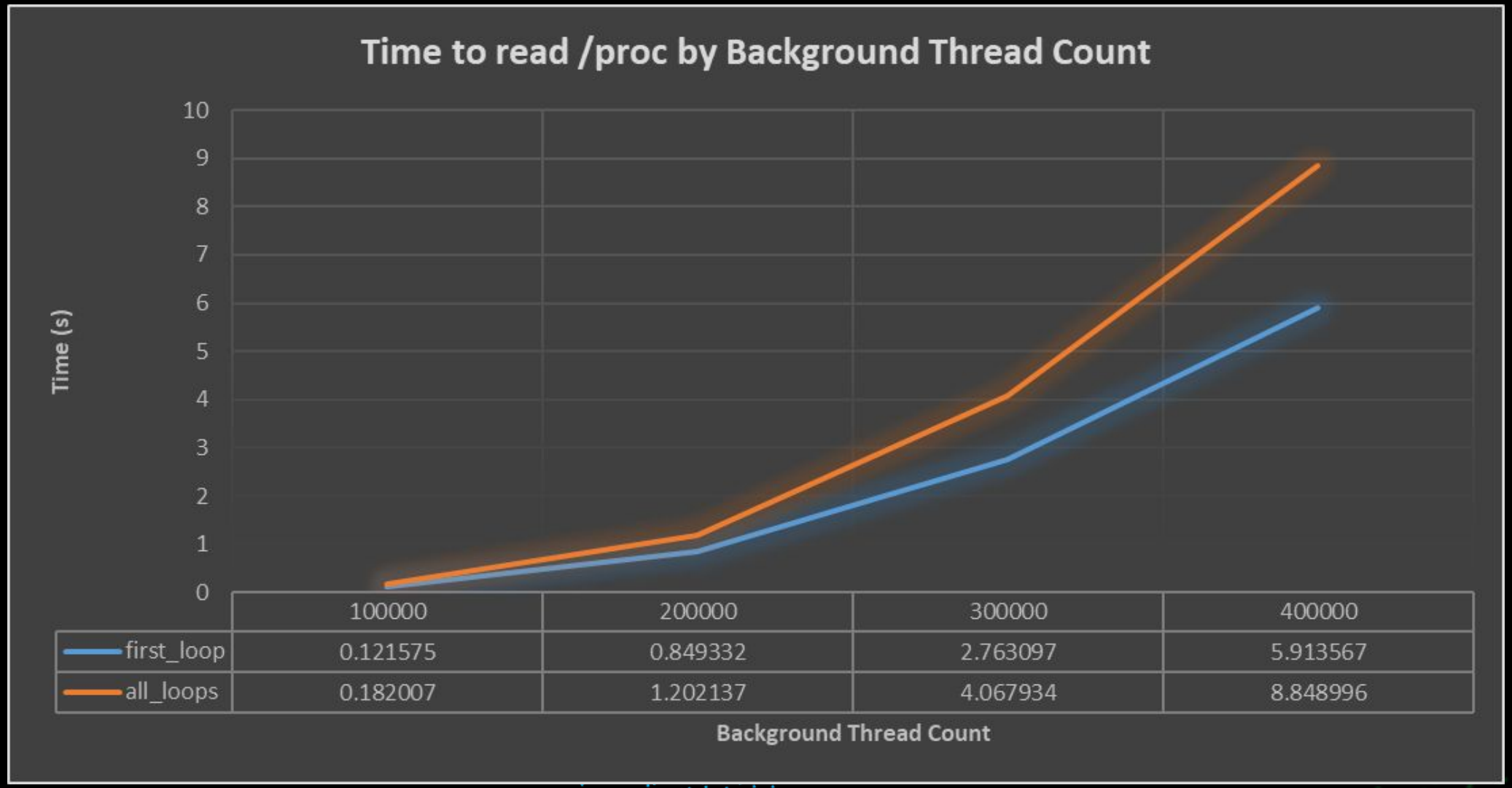
"The Prime Directive prohibits me from helping you."

# Profiling /proc

- The quote suggests contention should be mainly in *writers*
  - So far our data suggests its coming from the *readers*
  - Can we further support this?

- Where is the time spent?
  - Add timing to readdir call
    - After first loop
    - After all loops
  - Re-run thread chaos test in the background

# Profiling /proc




| | 100000 | 200000 | 300000 | 400000 |
|---|---|---|---|---|
| first_loop | 0.121575 | 0.849332 | 2.763097 | 5.913567 |
| all_loops | 0.182007 | 1.202137 | 4.067934 | 8.848996 |

# Profiling /proc – Summary

- Chaos testing told us our problems are correlated with high nproc and high read usage of /proc
- Used the /proc/profile system to profile kernel functions used to read /proc
- Further isolated to the set of functions responsible for building /proc/<pid> directories
- Data suggests contention on readers
  - Most time spent in first loop
    - Seems to be spent building the pid list - something done by readers
- But, design of RCU locks suggest contention on writers!

# Mitigation Efforts

- Chaos testing revealed our issues were tied to high nproc
- Potentially due to RCU lock-related contention in the kernel data structure used in `proc_pid_readdir`
  - pid namespaces
    - Containers
- CONFIG_RCU_FANOUT
  - “Lower fanout values reduce lock contention, but also consume more memory and increase the overhead of grace-period computations.” https://lwn.net/Articles/609904/#What%20Next%20for%20the%20RCU%20API
  - RCU locks are used for other kernel data structures – be careful!

# Mitigation Efforts

- Add nproc as a factor for load balancing
  - Contention looks exponential with respect to nproc
    - Reducing nproc by half gives ~4x returns
- Monitor and alert on threads
  - Per task and per host alerts

- Optimize applications for thread usage
  - Threads *were* cheap
    - Messaging libraries
    - Job queues
    - JVM
      - Default thread pools - Garbage Collection/JIT compiler
    - Review threading best practices

- Less of a problem in later kernel releases
  - Closer to linear growth with thread counts

# Summary

- Observed several issues with scaling
- Designed experiments to explore several potential causes
- Experiments point to nproc/threads as the culprit
- Repeat experiments show similar results
- Kernel profiling confirmed scaling issues
- Further investigation led to kernel /proc readdir related functions
- Now able to implement mitigation techniques

# On to the Next Experiment

- Chaos testing is a great proactive tool
- Also useful as an investigative tool
- What types problems could we use this technique on?
  - Difficult to reproduce
  - Wide ranging (more samples)
  - Regression testing

# Special Thanks

- TS SRE Runtime Team & OMS Team
- Bloomberg Resilience Engineering Team

TechAtBloomberg.com



Bloomberg

Engineering

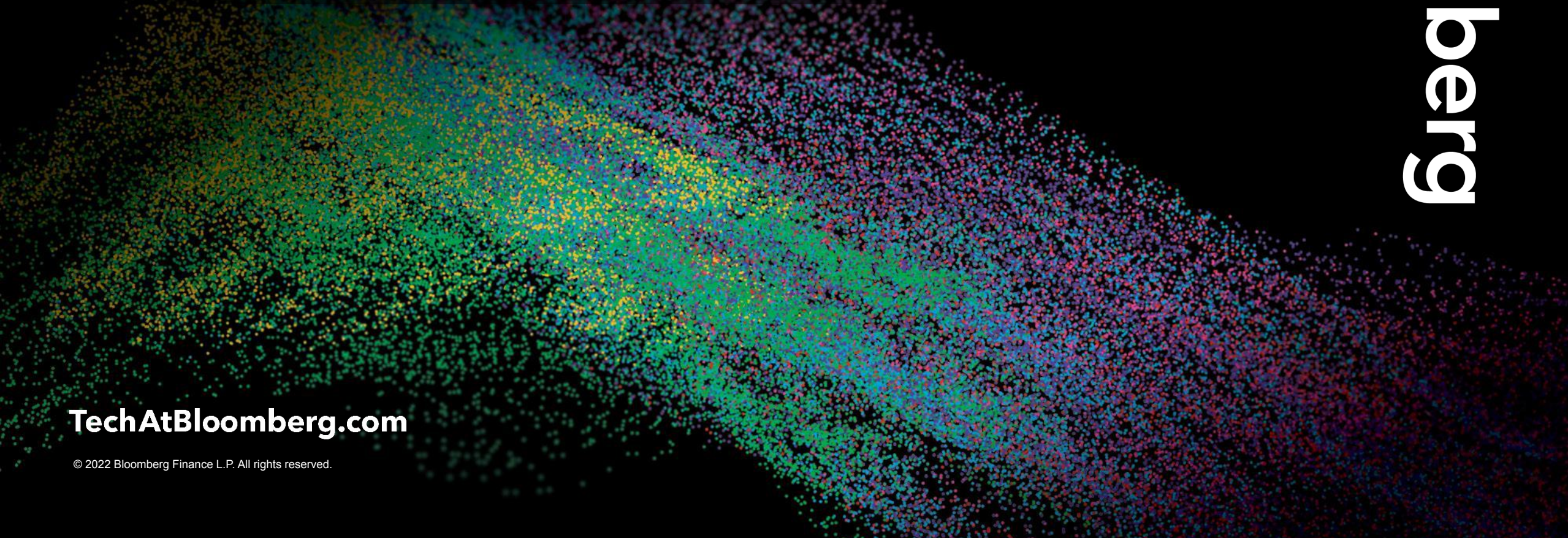
# Thank you!

https://TechAtBloomberg.com/blog/bloomberg-bets-big-on-sres/

https://www.bloomberg.com/careers

Bloomberg

Engineering



TechAtBloomberg.com

# Profiling /proc

- /proc/profile & readprofile (why not perf - doesn't work well on our hosts)
- Increments counter on clock tick

```
# readprofile | sort -n | tail -n 10
    16 get_signal_to_deliver                0.0106
    20 tick_nohz_idle_exit                  0.0595
    24 avtab_search_node                    0.1667
    35 __do_page_fault                      0.0277
    55 do_exit                              0.0210
    64 next_tgid                            0.3636
    70 release_task                         0.0599
    77 finish_task_switch                   0.1719
 24385 native_safe_halt                   762.0312
 25299 total                                0.0032
```

# of clock ticks  function name  # of ticks / size

# Profiling /proc

- Read-Copy-Update (RCU) Locks
  - Used for data structure synchronization
  - Called a lock but actually “lock free” for readers
- Designed for read-mostly data
  - Writers cannot block readers
  - Readers do not synchronize
    - Very low overhead
    - Rcu_read_lock is a no-op
      - Except when kernel allows preemption
  - Writers are blocked until readers are finished
- Kernel config
  - CONFIG_RCU_FANOUT
  - CONFIG_RCU_FANOUT_LEAF