Latency Distributions + Micro-benchmarks = Insights into Kernel Hotspots

SREcon21
October 12, 2021

Danny Chen
Trading Solutions SRE Team
dchen294@bloomberg.net
TechAtBloomberg.com
Biography

- UNIX performance engineer since 1980
- Worked on UNIX SVR3 and SVR4 virtual memory and demand paging
- Co-developed the first general purpose UNIX kernel tracing package
- Participated in the Performance Management Working Group - an industry-wide performance management standards effort
- Low latency market data
- Messaging and distributed transactions management
- Enterprise systems monitoring and capacity planning
- Working to get more “engineering” back in performance engineering
  - Visibility into Loggers… (SREcon19)
  - Pardon the Interposition… (LISA19)
  - Page Reference Sampling… (SREcon20)
Why Large Bare Metal Boxes?

- Faster local communication
  - UNIX Domain Sockets
  - Shared Memory
- Shared local state
- Assured durability of filesystem writes
- Control over resource allocation
  - High Volume and Low Latency Market Data
  - Real-time and near real-time requirements
The Scale in our Department

- >400K processes across hundreds of physical machines
  - 3 different platforms/operating systems (Linux, Solaris, AIX)
- 5-8K processes on busier hosts
- >250K threads on busier hosts
Case #1: SysV semaphore bottleneck (AIX)

- General system slowness on one of our production machines
  - Migrating services between machines did not help
    - Start-up scripts timed out
- Narrowing down the problem
  - Many services and utilities were slow
  - Using “trace” on one utility pointed to sporadic slow sem_init and sem_destroy times
Case #1: SysV semaphore bottleneck (AIX)

The micro-benchmark

```c
/* sema_load.c */
for (i = 0; i < n; i++) {
    sem_init(&sem[i], 0, 10);
    sem_destroy(&sem[i]);
}
```

Timings

```
$ time ./sema_load 3000000
real    0m8.274s
user    0m0.261s
sys     0m4.738s
```

<table>
<thead>
<tr>
<th>#</th>
<th>Avg. Wall</th>
<th>Avg. System</th>
<th>Avg. User</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.274</td>
<td>4.738</td>
<td>0.261</td>
</tr>
<tr>
<td>2</td>
<td>14.264</td>
<td>8.575</td>
<td>0.269</td>
</tr>
<tr>
<td>3</td>
<td>16.527</td>
<td>9.634</td>
<td>0.271</td>
</tr>
<tr>
<td>4</td>
<td>22.363</td>
<td>13.472</td>
<td>0.275</td>
</tr>
</tbody>
</table>
Case #1: Observations and Findings

- AIX CPU measurement when hyper-threading is very misleading
- No “out of the box” metrics on SysV IPC operations
  - Sporadic slowness (depending on concurrency/contention)
  - Took days to isolate the problem down to sem_init() and sem_destroy() operations
- sem_init() and sem_destroy() have critical regions that are protected by spin locks
  - Good for low contention
  - Bad during high contention
Case #2: SysV shared memory bottleneck (Linux)

● Low-level application infrastructure code dropping messages
  ○ Messaging leverages a form of “zero copy” IPC using SysV shared memory + message queues
  ○ What was causing “slow consumers”?
    ■ Application code?
    ■ Slow message queues?
    ■ Slow shared memory?

● Zeroing in on the problem
  ○ The “zero copy” mechanism puts out warnings when shmat() latency exceeds a threshold
Case #2: SysV shared memory bottleneck (Linux RHEL 6)

The micro-benchmark

```c
for (i = 0; i < numloops; i++) {
    void *vaddr = shmat(shmid, NULL, 0);
    shmdt(vaddr);
}
```

Timings

<table>
<thead>
<tr>
<th></th>
<th>Avg. Wall</th>
<th>Avg. System</th>
<th>Avg. User</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.235</td>
<td>2.344</td>
<td>0.061</td>
</tr>
<tr>
<td>4</td>
<td>98.809</td>
<td>33.587</td>
<td>1.580</td>
</tr>
</tbody>
</table>

$ time ./shm_load 3000000
real  0m3.235s
user  0m0.061s
sys   0m2.344s
Case #2: Observations and Findings

- No “out of the box” metrics on SysV IPC operations
  - Fortunately, the sub-system has measurements of the shmat/shmdt system calls
  - With logs upon crossing some threshold
- shmat() and shmdt() have critical regions that are protected by spin locks
  - Good for low contention
  - Bad during high contention
- Different in RHEL 7
  - Worse in 7.4
  - Much better in 7.6
Case #3: UNIX domain socket bottleneck (Solaris)

- Critical software infrastructure experiencing timeouts on load
  - Identity management with very strict SLOs
- Narrowing down the problem
  - A key SLI for the service is token generation latency
An Aside: Histograms and Distributions are Useful!

- More representative of the data set
  - Most data is not “normally distributed” -> means and std dev are not meaningful (and worse, misleading)
  - Is data bi-modal (or multi-modal)?
  - Long tails are meaningful
    - Sensitive detection of performance hiccups
  - Relatively compact storage requirements
  - Many SLAs and SLOs are stated in terms of distributions
An Aside: A Histogram Example

Latency Distribution (< 300 usec)

Latency Distribution (log scale)

~3300 usec
Case #3: Early Observations

● No “out of the box” metrics on socket operations
  ○ Fortunately, the sub-system kept distribution metrics on key latencies
  ○ This allowed an exact correlation between latency blips and execution of the netstat command

● The maximum netstat impact on latency varied widely from system to system
  ○ Conjecture: the level of impact was related to the number of UDS sockets on a system
    ■ Netstat holds a lock for the duration of its “read-only” operation when extracting the list of active UDS sockets
Case #3: UNIX domain socket bottleneck (Solaris)

The micro-benchmark #1 - testing against size

```c
#define MAX_TESTFDS 32*1024
for (i = 0; i < MAX_TESTFDS; i++) {
    fd[i] = socket(AF_UNIX, SOCK_STREAM, 0);
} pause();
```

Timings (sequential)

<table>
<thead>
<tr>
<th>#</th>
<th>Avg. Wall (sec)</th>
<th>Avg. System (sec)</th>
<th>Avg. User (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.240</td>
<td>.230</td>
<td>.011</td>
</tr>
<tr>
<td>2</td>
<td>.308</td>
<td>.298</td>
<td>.011</td>
</tr>
<tr>
<td>3</td>
<td>.371</td>
<td>.360</td>
<td>.011</td>
</tr>
<tr>
<td>4</td>
<td>.445</td>
<td>.443</td>
<td>.011</td>
</tr>
<tr>
<td>5</td>
<td>.552</td>
<td>.512</td>
<td>.011</td>
</tr>
<tr>
<td>6</td>
<td>.585</td>
<td>.573</td>
<td>.011</td>
</tr>
</tbody>
</table>
Case #3: Conclusions

- Solaris 11.3 is limited to a max of 256K UDS sockets
- The more UDS sockets there are, the longer it takes to create new, unbound UDS sockets
Case #4: Task clone and exit bottleneck (Linux)

- Preliminary: Does task creation/deletion take longer with more threads?
  - On hosts with >250K threads, we start to see timeouts in start-up and shutdown
Case #4: Task clone and exit bottleneck (Linux)

The micro-benchmark

```c
void *hangaround(void *args) {
    pause();
    return NULL;
}

int main(int argc, char *argv[]) {
    for (i = 0; i < nthreads; i++) {
        pthread_create(&tid, &attr, hangaround, NULL);
    }
    pause();
}
```
Case #4: Task clone and exit bottleneck (Linux)

$ time (ps auxww | wc)
   2967   42706  500273
real    0m0.174s
user    0m0.038s
sys     0m0.137s

$ ./lotsathreads 125000 &
$ time (ps auxww | wc)
   2984   42896  487592
real    0m0.482s
user    0m0.032s
sys     0m0.450s

$ ./lotsathreads 125000 &
$ time (ps auxww | wc)
   3032   43784  500467
real    0m1.892s
user    0m0.026s
sys     0m1.212s

- Note the growth in system time with threads
- Similar growth in system time if we `ls /proc`
- Answer: processes and threads are tasks to the Linux kernel
Case #4: Task clone and exit bottleneck (Linux)

$ for i in {1..1}; do time (ps > /dev/null) & done
real  0m1.175s
user  0m0.015s
sys   0m1.058s

$ for i in {1..2}; do time (ps > /dev/null) & done
real  0m3.139s
user  0m0.014s
sys   0m1.360s
real  0m3.449s
user  0m0.015s
sys   0m1.753s

● Note the serialization around concurrent ps instances
● There doesn’t appear to be a huge spin lock that ps (/proc access) encounters
● But ps is only reading data. Why the serialization around concurrent reads?
  ○ Is it possible that /proc access might impact task create/destroy?
  ○ Can task create/destroy also impact one another?
Summary

● Systems are not infinitely scalable
  ○ No OS has a monopoly on scale problems
● Latency histograms provide key visibility into spotting problems early
● Think of the kernel and the system call interface as a privileged library
  ○ Micro-benchmarks can help zero in on kernel hotspots
    ■ Complementary with kernel lock/tracing tools
    ■ Small, compact tests are easy to re-run
    ■ Be aware of “designing to the benchmark”
  ● Latency histograms can help compare “before and after” behavior
More Summary (Plea to Kernel Folks)

● The Prime Directive of Monitoring: Non-interference
  ○ Design monitoring interfaces and utilities to interact as minimally as possible with the system being monitored
  ○ Design the kernel to facilitate passive monitoring

● More visibility!
  ○ Latency histograms (as full fledged, full-time metrics) are crucially important
    ■ System calls
    ■ Key lock acquisition and hold
  ● Take care in use of spin locks
References

• Jon Bentley’s “Performance Bugs”: https://youtu.be/89qiHoDjeDg
• The case for histograms:
  — How NOT to Measure Latency (Gil Tene):
    https://www.youtube.com/watch?v=IJ8ydluPFeU
  — Latency SLOs Done Right (Fred Moyer):
    https://www.usenix.org/conference/srecon19americas/presentation/moyer
Thank you!

We are hiring: bloomberg.com/engineering