Performance tracing with BPF

Presented by Dongxu Huang / Wenbo Zhang

Catalog

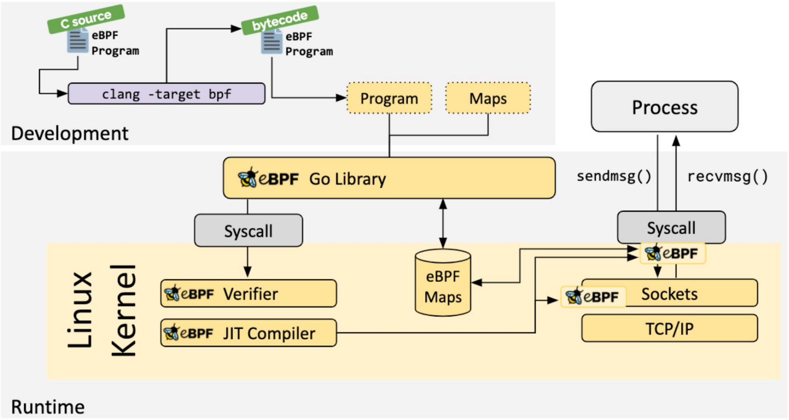
Part 1 - BPF's past and present
Part 2 - Why BPF?
Part 3 - Where to start?
Part 4 - Some real cases

Part 1 - BPF's past and present

What's BPF

It is a small stack-based virtual machine which runs programs injected from user space and attached to specific hooks in the kernel without changing kernel source code or loading kernel modules.

Fundamentally eBPF is still BPF, the Linux kernel community does not make a distinction, the official name is BPF. So we also follow this rule.



cBPF vs eBPF

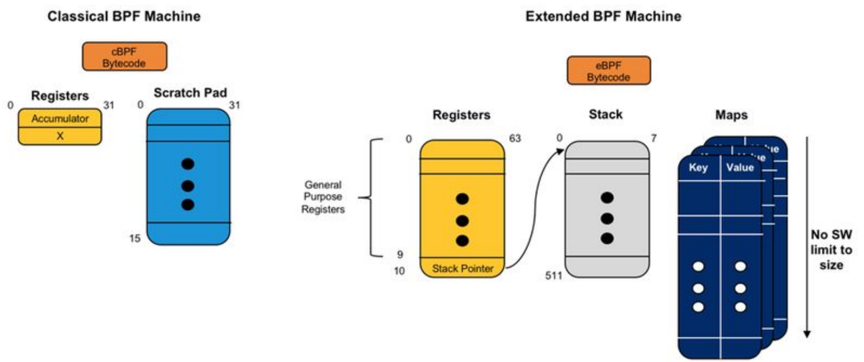
cBPF (legacy, Familiar and unfamiliar)

- a 32-bit wide accumulator, a 32-bit wide 'X' register which could also be used within instructions, and 16 32-bit registers which are used as a scratch memory store.

```
→  ~ sudo tcpdump -p -ni em1 -d "ip"
(000) ldh      [12]
(001) jeq      #0x800          jt 2    jf 3
(002) ret      #262144
(003) ret      #0
```

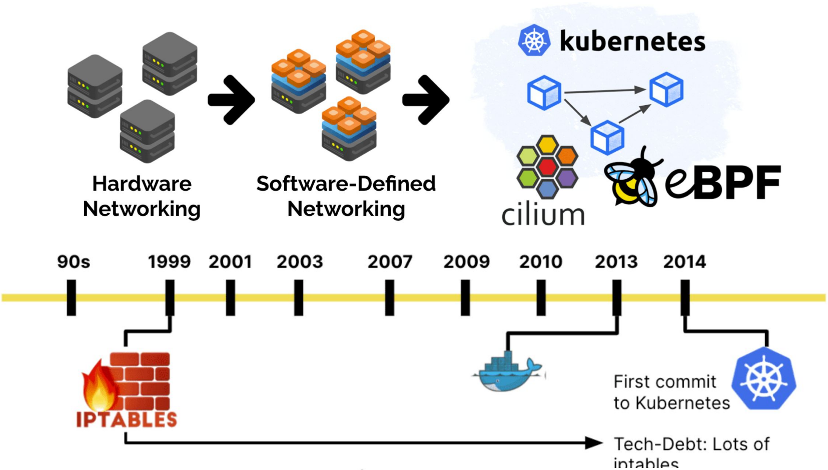
eBPF

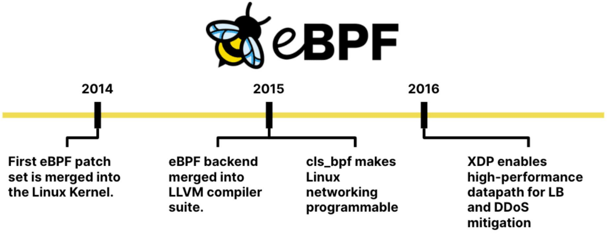
- an expanded set of registers and of instructions, the addition of maps (key/value stores without any restrictions in size), a 512 byte stack, more complex lookups, helper functions callable from inside the programs, and the possibility to chain several programs.



Major opportunity

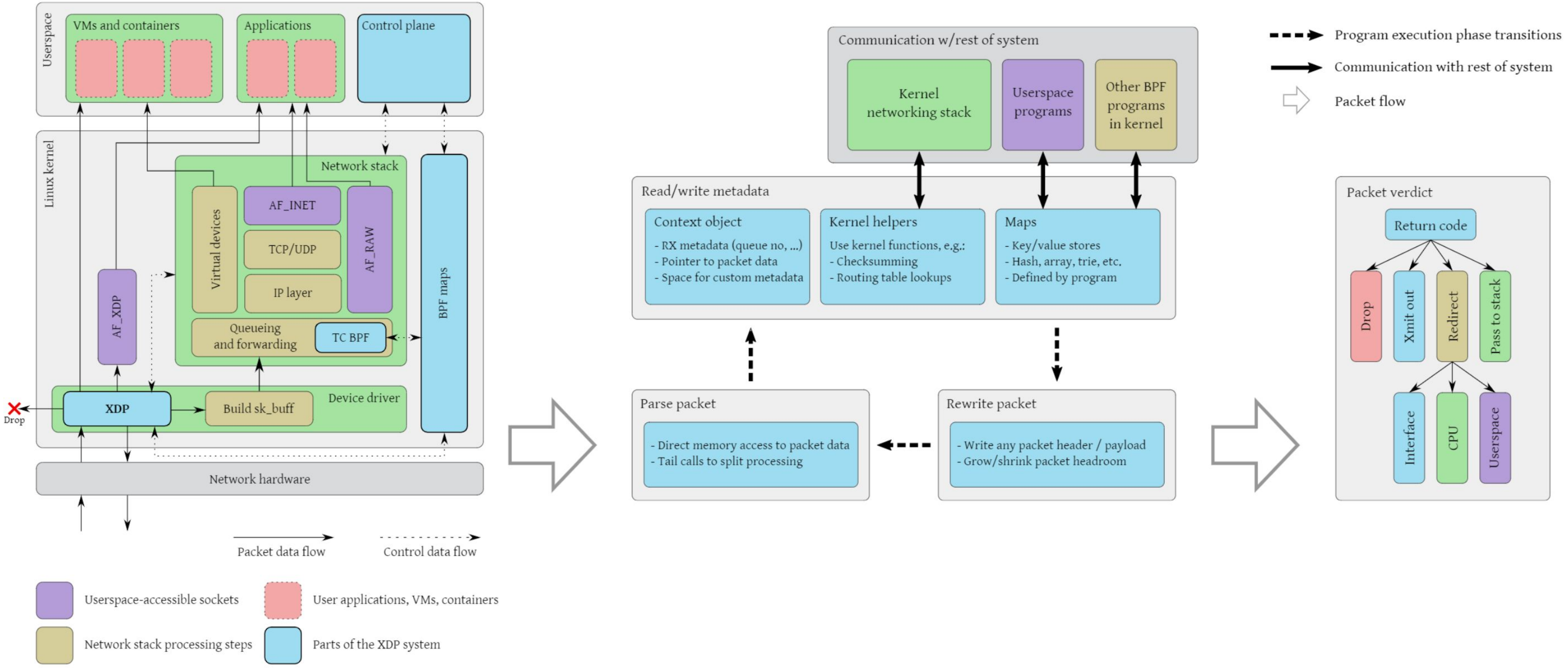
- Networking





Major opportunity

- Networking



Major opportunity

- Tracing

ftrace shortcomings:

- Plain text, sometimes unable to match data correctly or redundant

kprobe shortcomings:

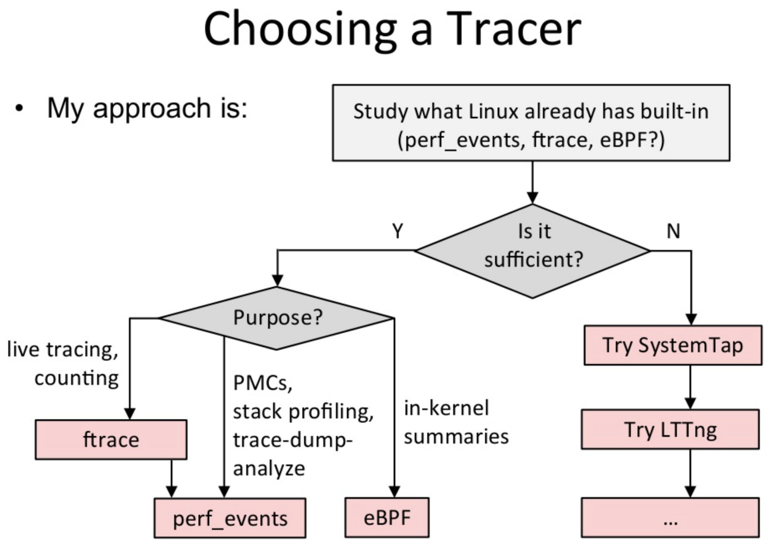
- Unsafe and poor compatibility may crash kernel

systemtap shortcomings:

- Uninstallation is not clean, causing kernel panic

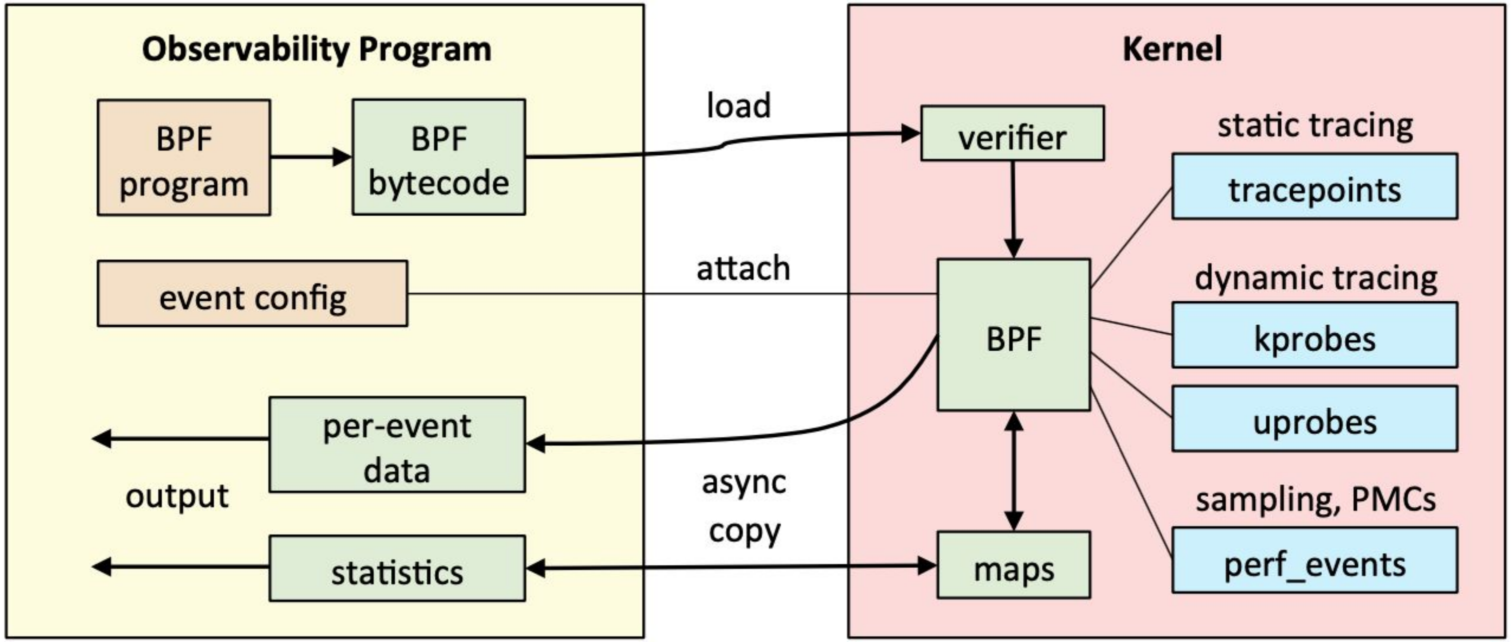
perf events

- Sometimes the overhead of record is too high



eBPF

BPF Tracing Internals



BPF dev experience —— Prehistoric

BPF Assembly

- CONS
  - Low development efficiency
  - inconvenient to deal with system calls directly

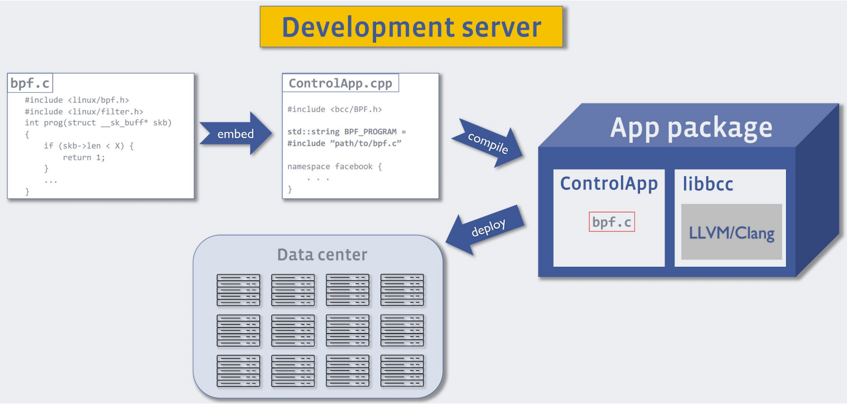
```
struct bpf_insn prog[] = {
        BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),
        BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol) /* R0 = ip->proto */),
        BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4), /* *(u32 *)(fp - 4) = r0 */
        BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
        BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4), /* r2 = fp - 4 */
        BPF_LD_MAP_FD(BPF_REG_1, map_fd),
        BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
        BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
        BPF_MOV64_IMM(BPF_REG_1, 1), /* r1 = 1 */
        BPF_ATOMIC_OP(BPF_DW, BPF_ADD, BPF_REG_0, BPF_REG_1, 0),
        BPF_MOV64_IMM(BPF_REG_0, 0), /* r0 = 0 */
        BPF_EXIT_INSN(),
};
size_t insns_cnt = sizeof(prog) / sizeof(struct bpf_insn);

prog_fd = bpf_load_program(BPF_PROG_TYPE_SOCKET_FILTER, prog, insns_cnt,
                           "GPL", 0, bpf_log_buf, BPF_LOG_BUF_SIZE);
```

BPF dev experience —— Big improve!

BPF Compiler Collection (BCC)

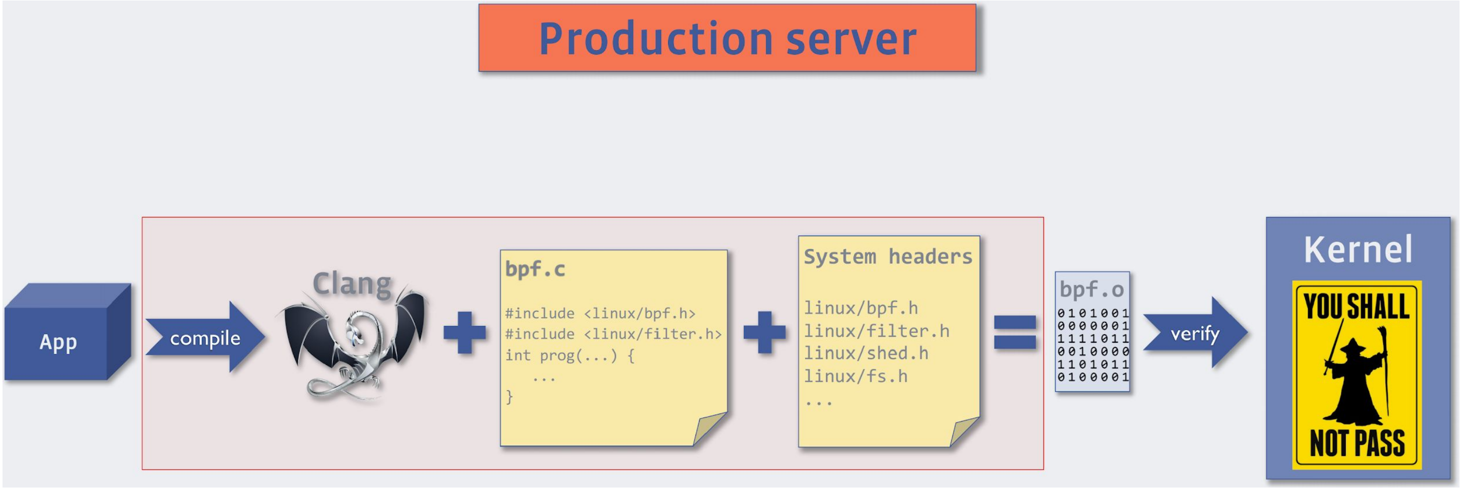
- PROS
  - Provide one-stop service
- CONS
  - Unnatural, need to remember some “magic”
  - Every machine needs to install kernel header packages
  - The libbcc library contains a huge LLVM or Clang library



BPF dev experience —— Next level !

libbpf + BPF CO-RE (Compile Once – Run Everywhere)

- PROS
  - Not much different from any “normal” user-space program
  - Smart BPF loader
  - Minimal dependencies



Current status of BPF

The rapid development of the community

- 31 BPF prog types
- 28 BPF map types
- New feature, such as
  - bpf spin lock
  - Kernel operations structures in BPF
  - Sleepable BPF programs
  - bpf iterator
  - bpf ring buffer
- More and more hook points
- More and more BPF helpers

Part 2 - Why BPF?

Why BPF?

If a desired behavior cannot be configured, a kernel change is required, historically, leaving two options:

- Native Support
  1. Change kernel source code and convince the Linux kernel community that the change is required.
  2. Wait several years for the new kernel version to become a commodity.

- Write a kernel module
  1. Fix it up regularly, as every kernel release may break it
  2. Risk corrupting your Linux kernel due to lack of security boundaries

A new option

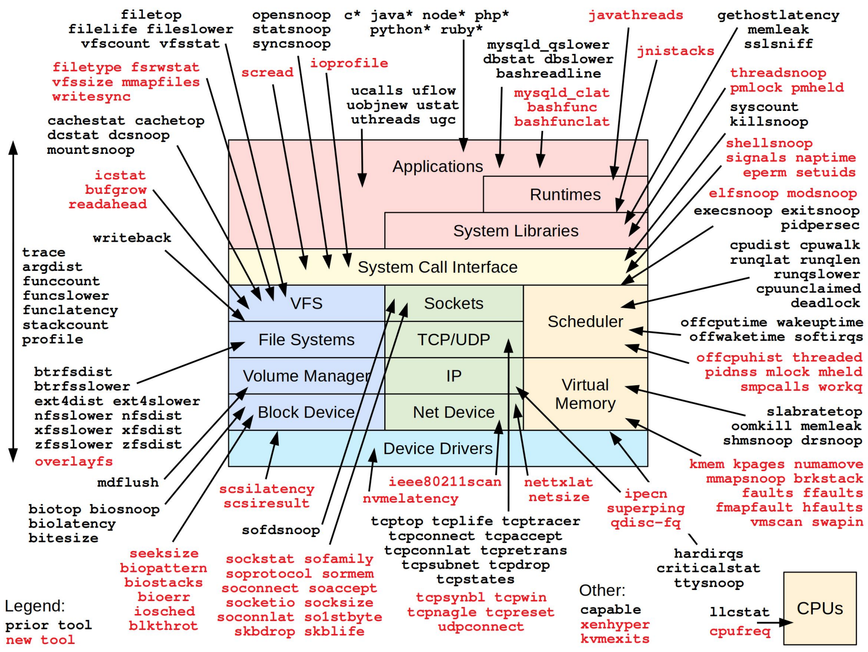
With BPF, a new option is available that allows for reprogramming the behavior of the Linux kernel without requiring changes to kernel source code or loading a kernel module. In many ways, this is very similar to how JavaScript and other scripting languages unlocked the evolution of systems which had become difficult or expensive to change.

Part 3 - Where to start?

Write a Hello world?

No, the best way to get started is to deploy and have fun

New tools developed for the book **BPF Performance Tools: Linux System and Application Observability**
by Brendan Gregg (Addison Wesley, 2019), which also covers **prior BPF tools**



Observe system calls

System call is the dividing line between user mode and kernel mode, observing system calls is the simplest entry point.

Think about these:

1. When the sys cpu usage rate is high, which system call on this machine has the most number of times?
2. When the sys cpu usage rate is high, which processes call a lot of system calls?
3. Which system calls take the longest time？
4. Does any system call return a special error value?
5. How to see the syscall's parameters and return value?

Observe system calls —— syscount

When the sys cpu usage rate is high, which system call on this machine has the most number of times?

```
# syscount
Tracing syscalls, printing top 10... Ctrl+C to quit.
[09:39:04]
SYSCALL                  COUNT
write                    10739
read                     10584
wait4                     1460
nanosleep                 1457
select                     795
rt_sigprocmask             689
clock_gettime              653
rt_sigaction               128
futex                       86
ioctl                       83
^C
```

Observe system calls —— syscount

When the sys cpu usage rate is high, which processes call a lot of system calls?

```
# syscount -P
Tracing syscalls, printing top 10... Ctrl+C to quit.
[09:58:13]
PID    COMM              COUNT
13820  vim                 548
30216  sshd                149
29633  bash                 72
25188  screen               70
25776  mysqld               30
31285  python               10
529    systemd-udevd         9
1      systemd               8
494    systemd-journal       5
^C
```

Observe system calls —— syscount

Which system calls take the longest time？

```
# syscount -L
Tracing syscalls, printing top 10... Ctrl+C to quit.
[09:41:32]
SYSCALL                   COUNT        TIME (us)
select                       16      3415860.022
nanosleep                   291        12038.707
ftruncate                     1          122.939
write                         4           63.389
stat                          1           23.431
fstat                         1            5.088
[unknown: 321]               32            4.965
timerfd_settime               1            4.830
ioctl                         3            4.802
kill                          1            4.342
^C
```

Observe system calls —— syscount

Does any system call return a special error value ?

```
# syscount -e ENOENT -i 5
Tracing syscalls, printing top 10... Ctrl+C to quit.
[13:15:57]
SYSCALL                    COUNT
stat                        4669
open                        1951
access                       561
lstat                         62
openat                        42
readlink                       8
execve                         4
newfstatat                     1

[13:16:02]
SYSCALL                    COUNT
lstat                      18506
stat                       13087
open                        2907
access                       412
openat                        19
readlink                      12
execve                         7
connect                        6
unlink                         1
rmdir                          1
^C
```

Observe system calls —— trace

How to see the syscall's parameters and return value?

```
# trace 'sys_execve "%s", arg1'
PID    COMM         FUNC            -
4402   bash         sys_execve      /usr/bin/man
4411   man          sys_execve      /usr/local/bin/less
4411   man          sys_execve      /usr/bin/less
4410   man          sys_execve      /usr/local/bin/nroff
4410   man          sys_execve      /usr/bin/nroff
4409   man          sys_execve      /usr/local/bin/tbl
4409   man          sys_execve      /usr/bin/tbl
4408   man          sys_execve      /usr/local/bin/preconv
4408   man          sys_execve      /usr/bin/preconv
4415   nroff        sys_execve      /usr/bin/locale
4416   nroff        sys_execve      /usr/bin/groff
4418   groff        sys_execve      /usr/bin/grotty
4417   groff        sys_execve      /usr/bin/troff
^C
```

```
# trace 'sys_read (arg3 > 20000) "read %d bytes", arg3'
PID    COMM         FUNC            -
4490   dd           sys_read        read 1048576 bytes
4490   dd           sys_read        read 1048576 bytes
4490   dd           sys_read        read 1048576 bytes
4490   dd           sys_read        read 1048576 bytes
^C
```

```
# trace.py -U -a 'r::sys_futex "%d", retval'
PID     TID     COMM            FUNC            -
793922  793951  poller          sys_futex       0
        7f6c72b6497a __lll_unlock_wake+0x1a [libpthread-2.23.so]
              627fef folly::FunctionScheduler::run()+0x46f [router]
        7f6c7345f171 execute_native_thread_routine+0x21 [libstdc++.so.6.0.21]
        7f6c72b5b7a9 start_thread+0xd9 [libpthread-2.23.so]
        7f6c7223fa7d clone+0x6d [libc-2.23.so]
```

What's next?

Try to play with:

- execsnoop
- opensoop
- tcplife
- ext4slower
- biosnoop

To observe system behavior

Stress test your system, or observe the online environment when there are problems such as resource shortage, find the right tool to analyze from the 150 tools

Develop new tools to meet your needs. If you think you can solve a type of problem, please submit it to the community. The community experts will not let you down

Use BPF in more areas, such as security, networking, etc.

Part 4 - Some real cases

Stealth process?

A 4-core virtual machine has a very high load. From the summary information of top, you can see that there are 10 tasks running, but only 1 task is running in the list.

What happened?

- These tasks wear stealth suits？
- Is there a bug in the Linux kernel?
- Is there a bug in top or pidstat?
- … ...

```
top - 14:32:58 up 17:35,  3 users,  load average: 8.91, 2.79, 1.45
Tasks: 156 total,  10 running, 143 sleeping,   0 stopped,   3 zombie
%Cpu(s): 32.6 us, 66.7 sy,  0.0 ni,  0.6 id,  0.0 wa,  0.0 hi,  0.1 si,  0.0 st
KiB Mem : 39122288 total, 38147032 free,   360832 used,   614424 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used. 38346216 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 3086 tidb      20   0  114164   2420   1208 R  97.7  0.0   0:41.07 bash
 5036 tidb      20   0  108364   1240    756 S   1.0  0.0   0:00.20 pidstat
    3 root      20   0       0      0      0 S   0.3  0.0   0:00.06 ksoftirqd/0
    7 root      rt   0       0      0      0 S   0.3  0.0   0:02.43 migration/0
   13 root      rt   0       0      0      0 S   0.3  0.0   0:00.70 migration/1
   18 root      rt   0       0      0      0 S   0.3  0.0   0:00.31 migration/2
   19 root      20   0       0      0      0 S   0.3  0.0   0:00.18 ksoftirqd/2
   23 root      rt   0       0      0      0 S   0.3  0.0   0:00.27 migration/3
   28 root      rt   0       0      0      0 S   0.3  0.0   0:00.14 migration/4
   29 root      20   0       0      0      0 S   0.3  0.0   0:00.13 ksoftirqd/4
   33 root      rt   0       0      0      0 S   0.3  0.0   0:00.12 migration/5
   43 root      rt   0       0      0      0 S   0.3  0.0   0:00.13 migration/7
   48 root      rt   0       0      0      0 S   0.3  0.0   0:02.04 migration/8
   49 root      20   0       0      0      0 S   0.3  0.0   0:00.06 ksoftirqd/8
   53 root      rt   0       0      0      0 S   0.3  0.0   0:02.89 migration/9
  285 root      20   0       0      0      0 S   0.3  0.0   0:00.01 kworker/8:1
    1 root      20   0   45976   6172   3848 S   0.0  0.0   0:01.94 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    4 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    6 root      20   0       0      0      0 S   0.0  0.0   0:00.02 kworker/u20:0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      20   0       0      0      0 S   0.0  0.0   0:01.60 rcu_sched
```

```
[tidb@localhost ~]$ pidstat 3
Linux 3.10.0-862.el7.x86_64 (localhost)       08/07/2019      _x86_64_       (10 CPU)

02:34:49 PM   UID       PID    %usr %system  %guest    %CPU   CPU  Command
02:34:52 PM     0         7    0.00    0.33    0.00    0.33     0  migration/0
02:34:52 PM     0        13    0.00    0.33    0.00    0.33     1  migration/1
02:34:52 PM     0        14    0.00    0.33    0.00    0.33     1  ksoftirqd/1
02:34:52 PM     0        18    0.00    0.33    0.00    0.33     2  migration/2
02:34:52 PM     0        19    0.00    0.33    0.00    0.33     2  ksoftirqd/2
02:34:52 PM     0        28    0.00    0.33    0.00    0.33     4  migration/4
02:34:52 PM     0        29    0.00    0.33    0.00    0.33     4  ksoftirqd/4
02:34:52 PM     0        33    0.00    0.33    0.00    0.33     5  migration/5
02:34:52 PM     0        38    0.00    0.33    0.00    0.33     6  migration/6
02:34:52 PM     0        39    0.00    0.33    0.00    0.33     6  ksoftirqd/6
02:34:52 PM     0        43    0.00    0.33    0.00    0.33     7  migration/7
02:34:52 PM     0        53    0.00    0.33    0.00    0.33     9  migration/9
02:34:52 PM  1000      3086   57.48   42.52    0.00  100.00     8  bash
```

Stealth process?

This prototype is based on the problem of abnormal cpu usage that my internal colleagues encountered when doing grpc-cpp vs grpc-rust bench in 2019 year. The environment at that time was also a little bit more complicated, in the docker . So maybe docker's bug? cgroup's bug? ...

```
[root@localhost perf-tools]# ./bin/execsnoop -d 1
Tracing exec()s for 1 seconds (buffered)...
Instrumenting sys_execve
   PID   PPID ARGS
 26382      0 ./a
 26383      0 ls -l /sys/kerne/debug
 26384      0 ./a
 26387   3086 nohup ./a
 26391  26382 <...> [?]
 26392  26388 sleep 1
 26389  25797 awk -v o=1 -v opt_name=0 -v name= -v opt_duration=1 [...]
 26385      0 ls -l /sys/kerne/debug
 26386      0 ls -l /sys/kerne/debug
 26390   3086 nohup ./a
 26394  26384 <...> [?]
 26393   3086 nohup ./a
 26396  26387 <...> [?]
 26395   3086 nohup ./a
 26397   3086 nohup ./a
 26399  26390 <...> [?]
 26400  26393 <...> [?]
 26398   3086 nohup ./a
 26401   3086 nohup ./a
 26402  26395 <...> [?]
 26404  26397 <...> [?]
 26406  26398 <...> [?]
 26405   3086 nohup ./a
 26407   3086 nohup ./a
 26403   3086 nohup ./a
 26408   3086 nohup ./a
 26409   3086 nohup ./a
 26410  26401 <...> [?]
 26411  26407 <...> [?]
 26413  26403 <...> [?]
 26412   3086 nohup ./a
 26414   3086 nohup ./a
 26416  26409 <...> [?]
 26415   3086 nohup ./a
 26418  26412 <...> [?]
```

Why does the process always sleep？

offcputime

```
# offcputime 5
Tracing off-CPU time (us) of all threads by user + kernel stack for 5 secs.

[...]
    finish_task_switch
    schedule
    schedule_timeout
    wait_woken
    sk_stream_wait_memory
    tcp_sendmsg_locked
    tcp_sendmsg
    inet_sendmsg
    sock_sendmsg
    sock_write_iter
    new_sync_write
    __vfs_write
    vfs_write
    SyS_write
    do_syscall_64
    entry_SYSCALL_64_after_hwframe
    __write
    [unknown]
    -                iperf (14657)
        5625
```

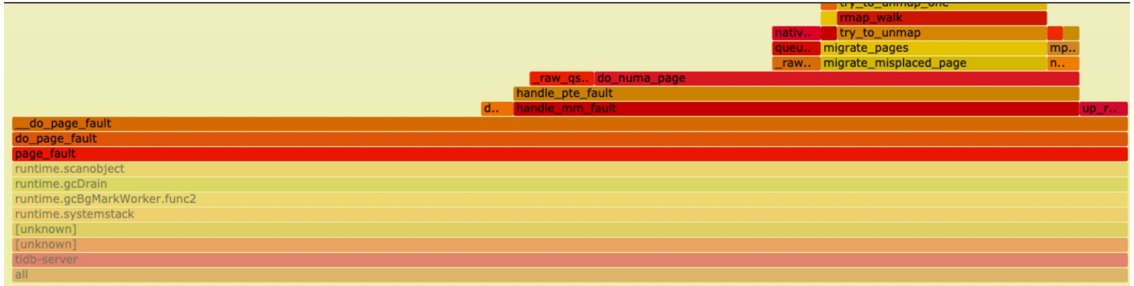
Weird minor page fault

One of our customers deployed TiDB in a virtual machine, which also has NUMA nodes. Because the number of CPUs is small, we did not bind cores. High-latency GC problems occur during business peaks, and there is no shortage of system resources from the monitoring. There is only one anomaly —— minor page faults. So what is going on?

Weird minor page fault

We see that sys cpu has a relatively high proportion, and it is relatively simple to analyze the problem of on-cpu. Look directly at the on-cpu flame graph.

If you are familiar with the principle of the NUMA scheduler, you can know from the flame graph that it is caused by autonuma.




```
# numamove.bt
Attaching 4 probes...
TIME            NUMA_migrations  NUMA_migrations_ms
22:48:45                      0                   0
22:48:46                      0                   0
22:48:47                    308                  29
22:48:48                      2                   0
22:48:49                      0                   0
22:48:50                      1                   0
22:48:51                      1                   0
[...]
```

Another weird minor page fault

A colleague of us found that when testing a tikv cluster on a cloud platform, there would be a problem of 99% delay of read IO request doubled. From the monitoring, we observe that when the delay is abnormal, the percentage of sys cpu usage will increase, and the corresponding minor page fault also looks abnormally high. We know the speed of the cloud disk is also very slow, so is it a disk problem? But if it is a disk reading, shouldn't it be a major page fault? Why are major page faults almost 0 and all minor page faults?

Another weird minor page fault

Let's sort out our thoughts

- The cloud host is a single node, so there is no previous autonuma problem
- From `sar -B` we see a lot of direct reclaim events
- TiKV allocates physical memory when reading the IO path

OK, let's see which path is allocating physical memory first (with stackcount):

Can minor page fault happened in kernel mode?

```
__alloc_pages_nodemask
handle_mm_fault
__do_page_fault
do_page_fault
page_fault
do_generic_file_read.constprop.52
generic_file_aio_read2
ext4_file_read
do_sync_read
vfs_read
sys_pread64
system_call_fastpath
__pread_nocancel
__libc_start_main
  279399
```

Another weird minor page fault

Can minor page fault happened in kernel mode? —— Yes, of course

In our case it is due to `copy_to_user`.

```
ssize_t pread(int fd, void *buf, size_t count,
off_t offset);
```

Like the defensive programming we did when designing the interface, the kernel does not trust the user address space passed by the user mode, so it will do some checks and make sure that the user space to be operated has been mapped to physical memory. If not, then deal with it.

Another question, TiKV uses a memory pool. The `buf` passed to `pread64` also comes from the memory pool, and swap is not used. Why do we need to allocate physical memory?

Another weird minor page fault

Another question, TiKV uses a memory pool. The `buf` passed to `pread64` also comes from the memory pool, and swap is not used. Why do we need to allocate physical memory?

Because the existence of the memory pool is just to avoid frequently calling system calls to apply and merge VMA operations. Physical memory is delayed allocation.

If we expect that the physical memory corresponding to a segment of VMA will not be reclaimed, we can use the `mlock` system call to put the page in a special LRU list.

Another weird minor page fault

Generally, the speed of physical memory allocation is very fast, but when the memory resources are insufficient, slow memory allocation will be entered. Synchronous direct memory recovery is very slow, especially for the popular servers with hundreds of GB of memory. The overhead of traversing the LRU lists will be huge.

We can use drsnoop to analyze.

Notice: In a slow memory allocation path, direct reclaim may occur more than once, so sometimes oom killer cannot get the chance to run, the system looks hung up.

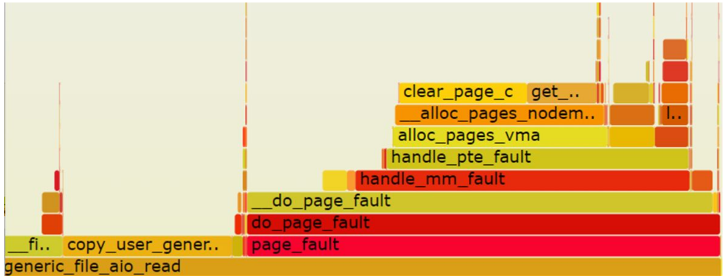
Linux kernel v4.12 limit max retry times to 16.

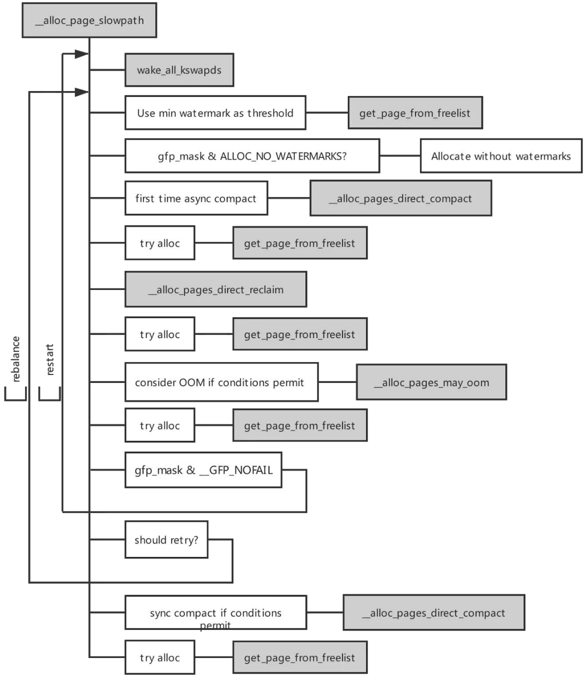
```
[centos@tao-tpcc-2 tools]$ sudo ./drsnoop.py
COMM             PID     LAT(ms) PAGES
sched-worker-p 2248        5.39  1568
unified-read-p 2248        6.54  1570
sched-worker-p 2248        6.01  1569
sched-worker-p 2248        6.55  1579
sched-worker-p 2248        6.72  1571
sched-worker-p 2248       14.73  2333
sched-worker-p 2248        7.75  2331
unified-read-p 2248       23.67  1570
sched-worker-p 2248       22.93  2367
unified-read-p 2248       20.76  1569
sched-worker-p 2248        3.91  1573
sched-worker-p 2248       14.31  1574
sched-worker-p 2248        5.79  2370
unified-read-p 2248        6.07  2341
unified-read-p 2248       12.51  2341
sched-worker-p 2248        4.50  1579
sched-worker-p 2248        5.71  1579
sched-worker-p 2248        8.47  1580
unified-read-p 2248        6.56  1589
unified-read-p 2248        6.07  1579
sched-worker-p 2248        6.73  2347
unified-read-p 2248        8.73  2348
sched-worker-p 2248       10.39  1580
```

Another weird minor page fault

So,
High sys cpu usages comes from `copy_user_generic_string`
High latency comes from `direct reclaim`

```
Samples: 499K of event 'cpu-clock', 4000 Hz, Event count (approx.): 103663569995 lost: 0/0 drop: 0/0
Overhead  Shared Object        Symbol
  15.88%  tikv-server          [.] rocksdb::crc32c::crc32c_3way
  12.90%  [kernel]             [k] copy_user_generic_string
   9.78%  [kernel]             [k] clear_page_c
   6.01%  [kernel]             [k] _raw_spin_unlock_irqrestore
   5.79%  [kernel]             [k] __do_page_fault
   5.29%  [kernel]             [k] get_page_from_freelist
   4.16%  [kernel]             [k] free_hot_cold_page
   2.81%  [kernel]             [k] __find_get_page
   2.57%  [kernel]             [k] unmap_page_range
   2.38%  [kernel]             [k] __mem_cgroup_commit_charge
   2.07%  [kernel]             [k] handle_mm_fault
   1.88%  [kernel]             [k] down_read_trylock
   1.67%  [kernel]             [k] up_read
   1.13%  tikv-server          [.] LZ4_compress_generic.constprop.5
```





Another weird minor page fault

Abstract into a simple example

```
➜  ~ vmtouch -vt README
README
[OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO] 5601/5601

           Files: 1
     Directories: 0
   Touched Pages: 5601 (21M)
         Elapsed: 0.008248 seconds
```

```
fd = open("./README", O_RDWR, 0644);
buf = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);

while (1) {
        if (pread(fd, buf, 4096, 0) < 0) {
                fprintf(stderr, "pread failed: %p, %s!\n", buf, strerror(errno));
                return -1;
        }
        madvise(buf, 4096, MADV_DONTNEED);
        usleep(1);
}
```

```
[root@localhost tools]# ./stackcount.py -p 30778 __alloc_pages_nodemask
Tracing 1 functions for "__alloc_pages_nodemask"... Hit Ctrl-C to end.
^C
  __alloc_pages_nodemask
  handle_mm_fault
  __do_page_fault
  do_page_fault
  page_fault
  do_generic_file_read.constprop.52
  generic_file_aio_read2
  ext4_file_read
  do_sync_read
  vfs_read
  sys_pread64
  system_call_fastpath
  __pread_nocancel
  __libc_start_main
    68688
```

```
[root@localhost tools]# ./stackcount.py -p 30896 __alloc_pages_nodemask
Tracing 1 functions for "__alloc_pages_nodemask"... Hit Ctrl-C to end.
^C
Detaching...
```

Premature memory reclamation?

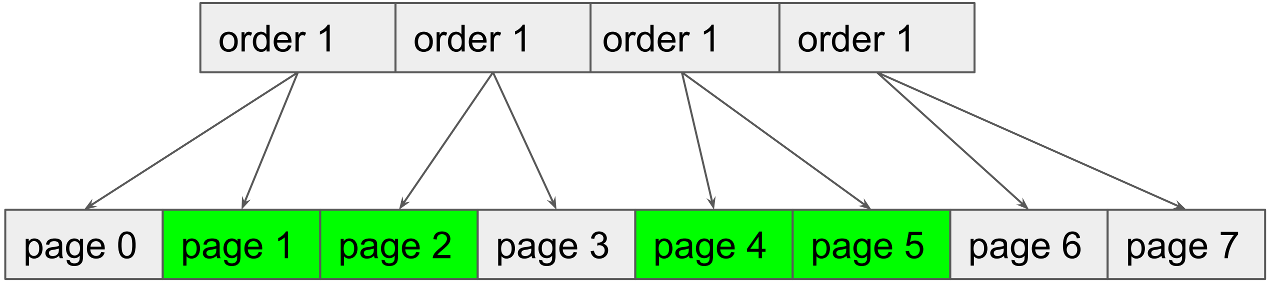
There are 3 TiKVs on a machine. After turning off 1 kv (to release the memory), the system stalling phenomenon disappears. It is confirmed by sar -B that it is related to the memory. When the machine stalls, there are direct reclaim events.

However, because the machine has a lot of free memory (> 20%), it has not reached the Linux memory recovery water mark. So is this a kernel bug?

Premature memory reclamation?

Let's remember linux physical memory management first:

- Buddy
- Linear mapping of kernel address space, allow allocation of high-order memory
- User space multi-level page table mapping



Are we buddy？

Buddy cond

- Two blocks of the same size
- Two block addresses are consecutive
- Two blocks must be separated from the same high order block（otherwise cavities）

Premature memory reclamation?

External memory fragmentation

```
<...>-46310 [005] .... 6403758.012379: mm_vmscan_direct_reclaim_begin: order=3 may_writepage=1
gfp_flags=GFP_TEMPORARY|GFP_NOWARN|GFP_NORETRY|GFP_COMP|GFP_NOTRACK
<...>-46310 [005] .... 6403758.012387: <stack trace>
=> alloc_pages_current
=> new_slab
=> ___slab_alloc
=> __slab_alloc
=> kmem_cache_alloc
=> proc_alloc_inode
=> alloc_inode
=> new_inode_pseudo
=> new_inode
=> proc_pid_make_inode
=> proc_fd_instantiate
=> proc_fill_cache
=> proc_readfd_common
=> proc_readfd
=> iterate_dir
=> SyS_getdents
```

```
[root@localhost ~]# cat /proc/buddyinfo
Node 0, zone      DMA      1      0      1      0      2      1      1      0      1      1      3
Node 0, zone    DMA32   1363   1772    266     58     81     78     44     41     35     11      6
Node 0, zone   Normal  19782     56      0      0      0      0      0      0      0      0      0
[root@localhost ~]# cat /proc/pagetypeinfo
Page block order: 9
Pages per block:  512

Free pages count per migrate type at order       0      1      2      3      4      5      6      7      8      9     10
Node    0, zone      DMA, type    Unmovable      1      0      1      0      2      1      1      0      1      0      0
Node    0, zone      DMA, type  Reclaimable      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone      DMA, type      Movable      0      0      0      0      0      0      0      0      0      1      3
Node    0, zone      DMA, type      Reserve      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone      DMA, type          CMA      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone      DMA, type      Isolate      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone    DMA32, type    Unmovable     35      2     26     46     80     77     43     41     34     11      0
Node    0, zone    DMA32, type  Reclaimable   1115    912      1      1      0      1      1      0      1      0      0
Node    0, zone    DMA32, type      Movable    351    779    236     15      1      0      0      0      0      0      6
Node    0, zone    DMA32, type      Reserve      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone    DMA32, type          CMA      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone    DMA32, type      Isolate      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone   Normal, type    Unmovable    464      0      0      0      0      0      0      0      0      0      0
Node    0, zone   Normal, type  Reclaimable  19222      8      0      0      0      0      0      0      0      0      0
Node    0, zone   Normal, type      Movable    139      6      0      0      0      0      0      0      0      0      0
Node    0, zone   Normal, type      Reserve      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone   Normal, type          CMA      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone   Normal, type      Isolate      0      0      0      0      0      0      0      0      0      0      0

Number of blocks type     Unmovable  Reclaimable      Movable      Reserve          CMA      Isolate
Node 0, zone      DMA            1            0            7            0            0            0
Node 0, zone    DMA32           58           38         1432            0            0            0
Node 0, zone   Normal          477          422        17097            0            0            0
[root@localhost ~]# cat /sys/kernel/debug/extfrag/
extfrag_index   unusable_index
[root@localhost ~]# cat /sys/kernel/debug/extfrag/extfrag_index
Node 0, zone      DMA -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
Node 0, zone    DMA32 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
Node 0, zone   Normal -1.000 -1.000 0.750 0.875 0.938 0.969 0.985 0.993 0.997 0.998 0.999
```

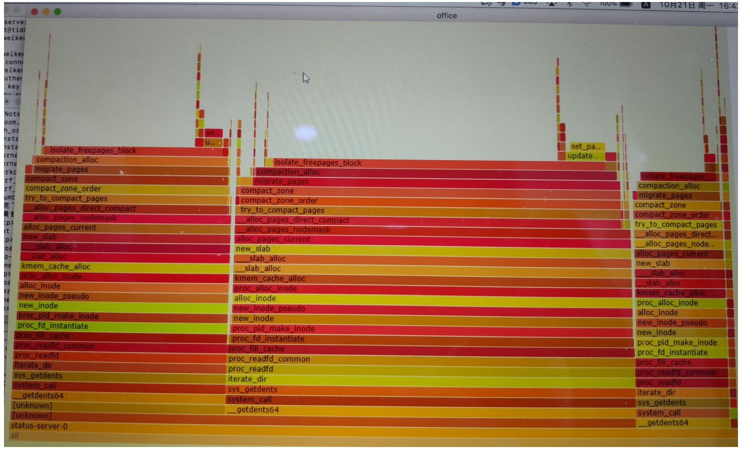
Premature memory reclamation?

Root issue

- Linux kernel design problem: In order to be simple and efficient, the kernel space adopts linear mapping and allows to apply for high-level physical memory. When the system runs for a long time and generates external memory fragments and cannot meet the high-level memory requirements, it will trigger direct memory recycling or regularization. The system has high latency or high sys cpu usage.

We can use these tools to analysis:

- [drsnoop]
- [compactsnoop]
- [trace]
- [profiler] + flame graph



Is read ahead aggressive?

This shows that during tracing there were 128 pages read ahead but unused (that's not many). The histogram shows thousands of pages were read and used, mostly within 32 milliseconds. If that time was in the many seconds, it could be a sign that read-ahead is loading too aggressively, and should be tuned.

```
# readahead.bt
Attaching 5 probes...
^C
Readahead unused pages: 128

Readahead used page age (ms):
@age_ms:
[1]                 2455 |@@@@@@@@@@@@@@@                                     |
[2, 4)              8424 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[4, 8)              4417 |@@@@@@@@@@@@@@@@@@@@@@@@@@@                         |
[8, 16)             7680 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     |
[16, 32)            4352 |@@@@@@@@@@@@@@@@@@@@@@@@@@                          |
[32, 64)               0 |                                                    |
[64, 128)              0 |                                                    |
[128, 256)           384 |@@                                                  |
```

When should we expand the memory?

For database applications, the page cache hit rate has a great impact on performance. When the remaining memory is insufficient and the working set size exceeds the current memory capacity, expansion needs to be considered. How to determine it?

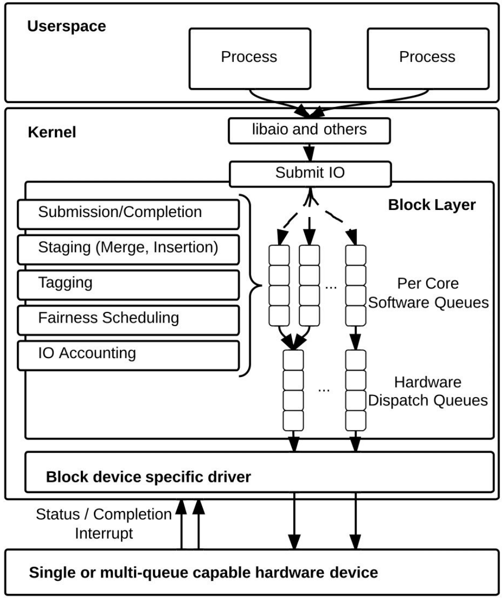
cachestat

```
# cachestat -T 10
TIME           HITS   MISSES  DIRTIES HITRATIO    BUFFERS_MB  CACHED_MB
21:08:58        771        0        1  100.00%             8        190
21:09:08      33036    53975       16   37.97%             9        400
21:09:18         15    68544        2    0.02%             9        668
21:09:28        798    65632        1    1.20%             9        924
21:09:38          5    67424        0    0.01%             9       1187
21:09:48       3757    11329        0   24.90%             9       1232
21:09:58       2082        0        1  100.00%             9       1232
21:10:08     268421       11       12  100.00%             9       1232
21:10:18          6        0        0  100.00%             9       1232
21:10:19        784        0        1  100.00%             9       1232
```

Wether slow IO is caused by disk or not?

The IO stack of Linux is deeper, including the file system layer, block layer, and driver layer. These layers are also affected by the memory subsystem. So when the IO is slow, how do we determine whether it is a slow disk?

```
# ./biolatency
Tracing block device I/O... Hit Ctrl-C to end.
^C
     usecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 0        |                                        |
         8 -> 15         : 0        |                                        |
        16 -> 31         : 0        |                                        |
        32 -> 63         : 0        |                                        |
        64 -> 127        : 1        |                                        |
       128 -> 255        : 12       |********                                |
       256 -> 511        : 15       |**********                              |
       512 -> 1023       : 43       |********************************        |
      1024 -> 2047       : 52       |****************************************|
      2048 -> 4095       : 47       |************************************    |
      4096 -> 8191       : 52       |****************************************|
      8192 -> 16383      : 36       |****************************            |
     16384 -> 32767      : 15       |**********                              |
     32768 -> 65535      : 2        |*                                       |
     65536 -> 131071     : 2        |*                                       |
```




```
# ./biosnoop
TIME(s)     COMM           PID    DISK    T  SECTOR     BYTES   LAT(ms)
0.000004    supervise      1950   xvda1   W  13092560   4096       0.74
0.000178    supervise      1950   xvda1   W  13092432   4096       0.61
0.001469    supervise      1956   xvda1   W  13092440   4096       1.24
0.001588    supervise      1956   xvda1   W  13115128   4096       1.09
```

How to get tcp retrans without capturing？

The pressure test tcp retrans is very high, is there a way to find which ip from the source end to which ip a large amount of retrans occurred without tcpdump?

```
# ./tcpretrans.py -c
Tracing retransmits ... Hit Ctrl-C to end
^C
LADDR:LPORT              RADDR:RPORT          RETRANSMITS
192.168.10.50:60366  <-> 172.217.21.194:443       700
192.168.10.50:666    <-> 172.213.11.195:443       345
192.168.10.50:366    <-> 172.212.22.194:443       211
[...]
```

Q & A

Thank you!