# Fast and Safe Production Monitoring of JVM Applications with BPF Magic

Sasha Goldshtein
CTO, Sela Group

goldshtn
goldshtn

# The Plan

- This is a talk on hardcore Linux tracing tools and how they can be used with JVM applications
  - For non-Linux platforms, rough equivalents are sort of available (e.g. dtrace on macOS and FreeBSD)
- You'll learn:

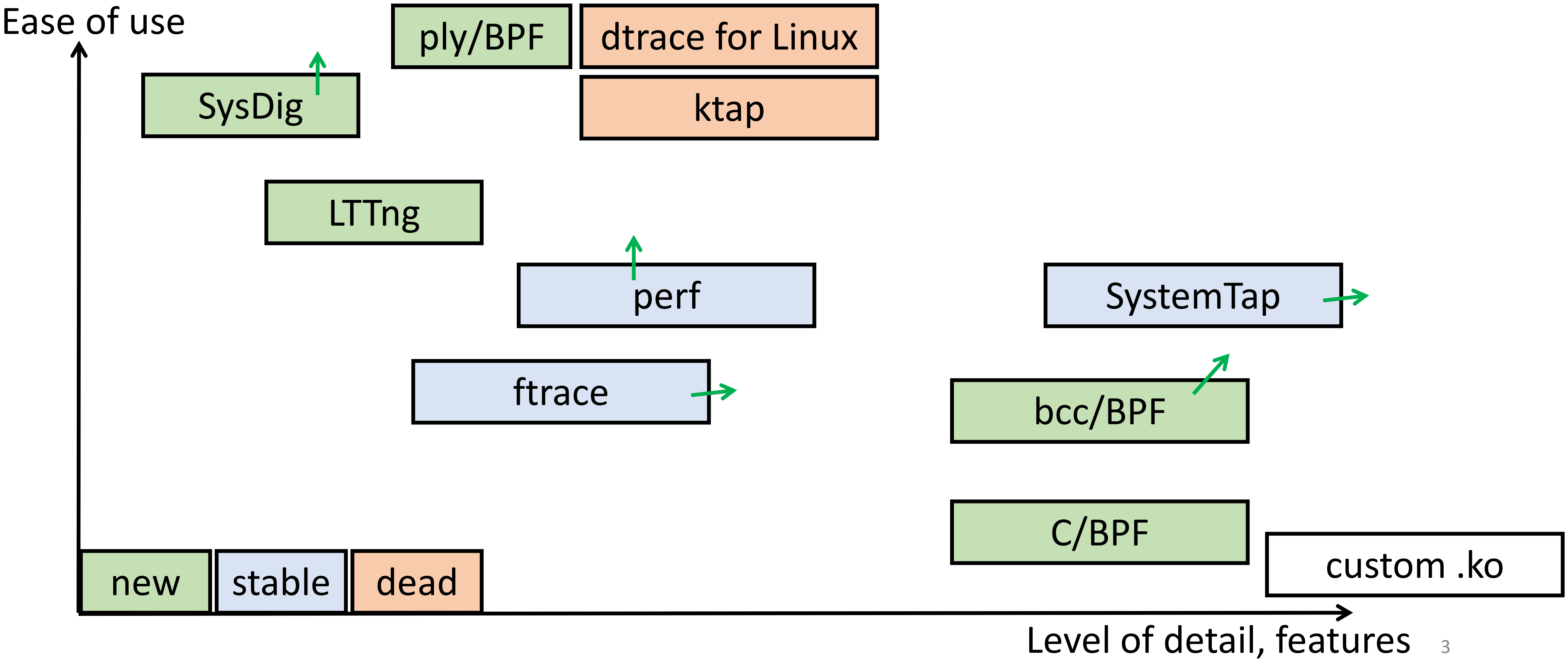
❑Which production-ready tracing tools can be used with JVM apps

❑How BPF changes the picture of Linux tracing

❑To apply a performance checklist for JVM apps using BPF tools

❑To conduct ad-hoc investigations with one-liners and custom tools

# Landscape of Linux Tracing Tools



# Demo: Observability Points in the JVM 👀

- Objective:
  Understand which tracepoints are available in the JVM

```
$ find /usr/lib/jvm -name libjvm.so -exec readelf -n {} + | grep -A2 NT_STAPSDT
  stapsdt              0x00000078       NT_STAPSDT (SystemTap probe descriptors)
    Provider: hotspot
    Name: mem__pool__gc__begin
--
  stapsdt              0x0000004d       NT_STAPSDT (SystemTap probe descriptors)
    Provider: hotspot
    Name: class__loaded
--
  stapsdt              0x0000004e       NT_STAPSDT (SystemTap probe descriptors)
    Provider: hotspot
    Name: object__alloc
--
  stapsdt              0x00000065       NT_STAPSDT (SystemTap probe descriptors)
    Provider: hotspot
    Name: thread__start
…
```

`readelf` displays the raw SDT notes from the binary, a quick way to identify which probes are available

```
$ tplist -p `pidof java`
/usr/lib/jvm/.../server/libjvm.so hotspot:object__alloc
/usr/lib/jvm/.../server/libjvm.so hotspot:method__entry
/usr/lib/jvm/.../server/libjvm.so hotspot:method__return
/usr/lib/jvm/.../server/libjvm.so hotspot:monitor__waited
/usr/lib/jvm/.../server/libjvm.so hotspot:monitor__wait
/usr/lib/jvm/.../server/libjvm.so hotspot:thread__stop
/usr/lib/jvm/.../server/libjvm.so hotspot:thread__start
/usr/lib/jvm/.../server/libjvm.so hotspot:vm__init__begin
/usr/lib/jvm/.../server/libjvm.so hotspot:vm__init__end
/usr/lib/jvm/.../server/libjvm.so hotspot:thread__unpark
/usr/lib/jvm/.../server/libjvm.so hotspot:thread__park__begin
/usr/lib/jvm/.../server/libjvm.so hotspot:thread__park__end
/usr/lib/jvm/.../server/libjvm.so hs_private:cms__initmark__begin
/usr/lib/jvm/.../server/libjvm.so hs_private:cms__initmark__end
/usr/lib/jvm/.../server/libjvm.so hs_private:cms__remark__begin
/usr/lib/jvm/.../server/libjvm.so hs_private:cms__remark__end
/usr/lib/jvm/.../server/libjvm.so hotspot:gc__begin
/usr/lib/jvm/.../server/libjvm.so hotspot:gc__end
/usr/lib/jvm/.../server/libjvm.so hotspot:vmops__begin
/usr/lib/jvm/.../server/libjvm.so hotspot:vmops__end
/usr/lib/jvm/.../server/libjvm.so hotspot:vmops__request
…
```

`tplist` (from BCC) can also display a list of probes from a binary or a running process

```
$ find /usr/lib/jvm -name libjvm.so -exec tplist -vv -l {} + | grep monitor__waited -A10
/usr/lib/jvm/.../server/libjvm.so hotspot:monitor__waited [sema 0x0]
  location #1 0xa0c4dd
    argument #1 8 signed   bytes @ ax
    argument #2 8 unsigned bytes @ r12
    argument #3 8 unsigned bytes @ dx
    argument #4 4 signed   bytes @ cx
  location #2 0xa0e85d
    argument #1 8 signed   bytes @ ax
    argument #2 8 unsigned bytes @ r15
    argument #3 8 unsigned bytes @ dx
    argument #4 4 signed   bytes @ cx
```

`tplist` can show which arguments are passed to the probes, but can't interpret them

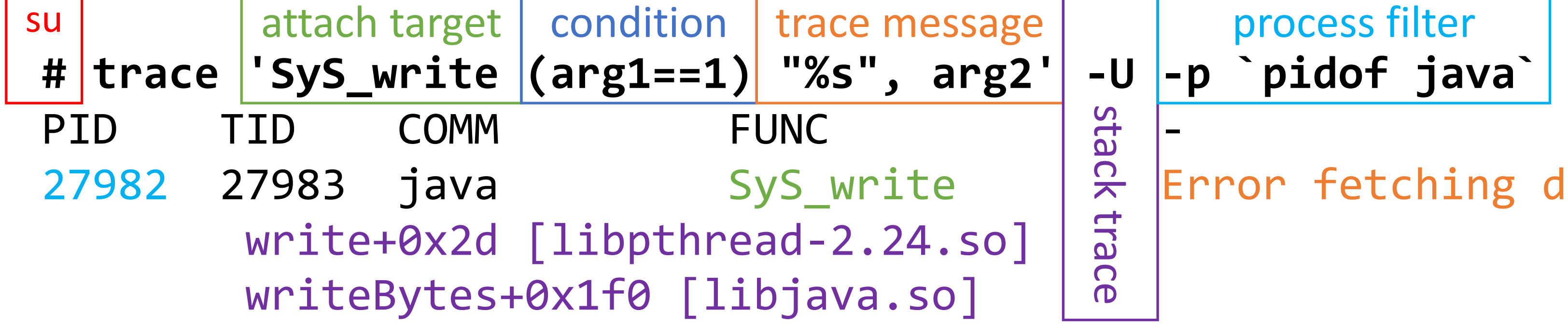
```
$ find /usr/lib/jvm -name hotspot-*.stp -exec grep 'mark("monitor__waited")' -A10 {} +
process("/usr/lib/jvm/.../server/libjvm.so").mark("monitor__waited")
{
  name = "monitor_waited";
  thread_id = $arg1;
  id = $arg2;
  class = user_string_n($arg3, $arg4);
  probestr = sprintf("%s(thread_id=%d,id=0x%x,class='%s')",
                     name, thread_id, id, class);
}
```

Some software ships with .stp or .d files that explain the probe structure

# Demo: Methods and Stack Traces 👀

- Objective:
  A Java app is printing undesired stuff to the console, and you want to understand where it's coming from
- JVM flags we will need:
  - `-XX:+PreserveFramePointer` ~3% overhead, helps get good stacks

```
$ java … myapp
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
Error fetching data, cleaning up.
```



```
# trace 'SyS_write (arg1==1) "%s", arg2' -U -p `pidof java`
PID    TID    COMM          FUNC             -
27982  27983  java          SyS_write        Error fetching data, cleaning up.0
        write+0x2d [libpthread-2.24.so]
        writeBytes+0x1f0 [libjava.so]
        Java_java_io_FileOutputStream_writeBytes+0x1a [libjava.so]
        [unknown] [perf-15527.map]
        [unknown] [perf-15527.map]
        [unknown] [perf-15527.map]
        [unknown] [perf-15527.map]
        [unknown] [perf-15527.map]
        [unknown] [perf-15527.map]
        [unknown] [perf-15527.map]
        [unknown] [perf-15527.map]
…
        JavaCalls::call_helper(JavaValue*, methodHandle*, …)+0xf53 [libjvm.so]
        jni_invoke_static(JNIEnv_*, JavaValue*, _jobject*, …)+0x357 [libjvm.so]
        jni_CallStaticVoidMethod+0x186 [libjvm.so]
        JavaMain+0x6d1 [libjli.so]
        start_thread+0xca [libpthread-2.24.so]
```

🤔

✂ trace (from BCC) attaches a dynamic trace to an arbitrary location; in this case, the `write` syscall

```
$ create-java-perf-map.sh `pidof java` "unfoldall,dottedclass"
$ tail /tmp/perf-`pidof java`.map
7f6ed52ea5c0 100 DataFetcher::processIt
7f6ed52eaa20 880 java.lang.ClassLoader::loadClass
7f6ed52ebf40 340 DataFetcher::fetchData
7f6ed52ec660 220 RequestProcessor::processRequest
7f6ed52ecbe0 220 RequestProcessor::processRequest
7f6ed52ed140 120 DataFetcher::<init>
7f6ed52ed500 180 sun.misc.URLClassPath$FileLoader$1::getInputStream
7f6ed52ed9c0 520 java.lang.ClassLoader::loadClass
7f6ed52eff20 100 DataFetcher::fetchData
7f6ed52f1e80 4c0 java.lang.ThreadGroup::add
```

`create-java-perf-map.sh` (from perf-map-agent) generates a symbol file describing Java symbols and addresses

```
# trace 'SyS_write (arg1==1) "%s", arg2' -U -p `pidof java`
PID    TID    COMM         FUNC             -
25335  25336  java         SyS_write        Error fetching data, cleaning up.f8
        write+0x24 [libpthread-2.24.so]
        writeBytes+0x1f0 [libjava.so]
        Java_java_io_FileOutputStream_writeBytes+0x1a [libjava.so]
        java.io.FileOutputStream::writeBytes+0xc6 [perf-15527.map]
        java.io.FileOutputStream::write+0x74 [perf-15527.map]
        java.io.BufferedOutputStream::flushBuffer+0xa5 [perf-15527.map]
        java.io.BufferedOutputStream::flush+0x98 [perf-15527.map]
        java.io.PrintStream::write+0xf8 [perf-15527.map]
        sun.nio.cs.StreamEncoder::writeBytes+0x13c [perf-15527.map]
        sun.nio.cs.StreamEncoder::implFlushBuffer+0xcc [perf-15527.map]
        sun.nio.cs.StreamEncoder::flushBuffer+0xb8 [perf-15527.map]
        java.io.OutputStreamWriter::flushBuffer+0x85 [perf-15527.map]
        java.io.PrintStream::newLine+0xf4 [perf-15527.map]
        java.io.PrintStream::println+0xb0 [perf-15527.map]
        DataFetcher::fetchData+0xd4 [perf-15527.map]
        RequestProcessor::processRequest+0xc0 [perf-15527.map]
        Collecty::main+0x68 [perf-15527.map]
        call_stub+0x88 [perf-15527.map]
        JavaCalls::call_helper(JavaValue*, methodHandle*, …)+0xf53 [libjvm.so]
…
```

😑

# Demo: Methods and Stack Traces 👀

- Objective:
  A Java app is causing a lot of garbage collections by invoking `System.gc()` directly, and you want to understand why
- JVM flags we will need:

| | |
|---|---|
| `-XX:+PreserveFramePointer` | ~3% overhead, helps get good stacks |
| `-XX:+ExtendedDTraceProbes` | very expensive, only for debugging method calls/object allocations |

```
$ java … -XX:+PrintGC myapp
[Full GC (System.gc())  530K->255K(15872K), 0.0021490 secs]
[Full GC (System.gc())  255K->255K(15936K), 0.0020310 secs]
[Full GC (System.gc())  255K->255K(15936K), 0.0017840 secs]
[Full GC (System.gc())  255K->253K(15936K), 0.0019176 secs]
[Full GC (System.gc())  254K->253K(15936K), 0.0018467 secs]
[Full GC (System.gc())  254K->253K(15936K), 0.0018358 secs]
```

attach target

condition

trace message

stack trace

process filter

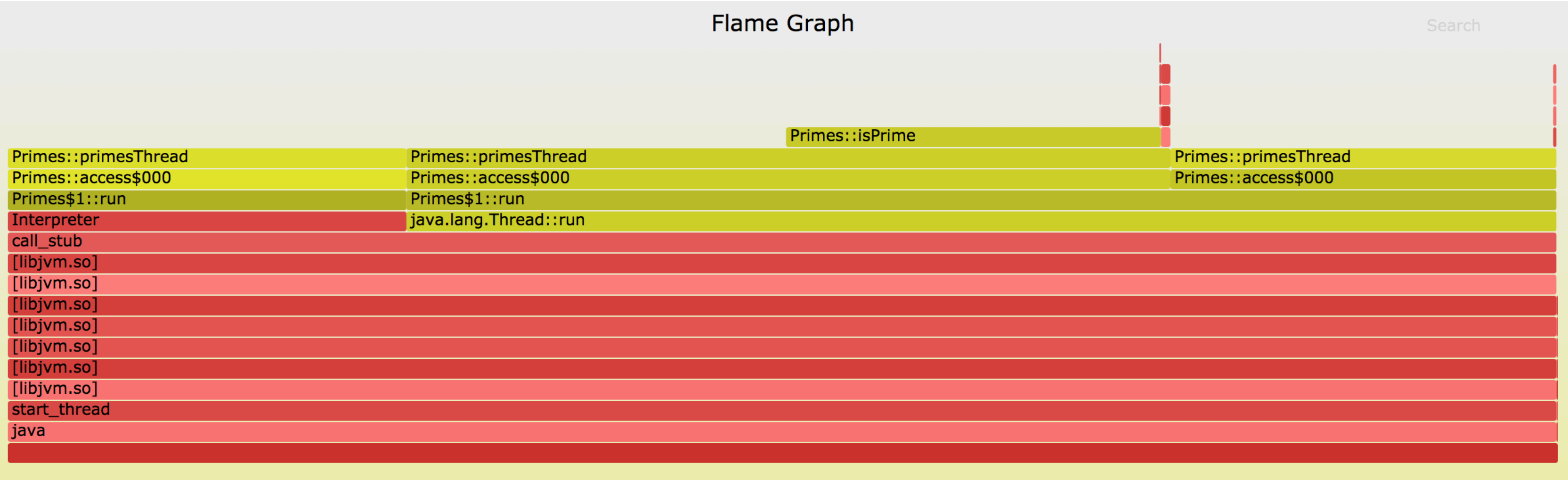
```
# trace 'u:libjvm.so:method__entry (STRCMP("gc", arg4)) "induced GC"' -U -p `pidof java`
PID    TID    COMM         FUNC            -
25413  25414  java         method__entry   induced GC
        SharedRuntime::dtrace_method_entry(…)+0x7b [libjvm.so]
        java.lang.Runtime::gc+0x80 [perf-15605.map]
        java.lang.System::gc+0x40 [perf-15605.map]
        DataFetcher::fetchData+0xdc [perf-15605.map]
        RequestProcessor::processRequest+0xc0 [perf-15605.map]
        Collecty::main+0x14b [perf-15605.map]
        call_stub+0x88 [perf-15605.map]
        JavaCalls::call_helper(JavaValue*, methodHandle*, …)+0xf53 [libjvm.so]
        jni_invoke_static(JNIEnv_*, JavaValue*, _jobject*, …)+0x357 [libjvm.so]
        jni_CallStaticVoidMethod+0x186 [libjvm.so]
        JavaMain+0x6d1 [libjli.so]
        start_thread+0xca [libpthread-2.24.so]
```

here, `trace` attaches to a relatively expensive method-entry probe that requires `-XX:+ExtendedDTraceProbes`

# OK, So You Have Probably Met **`perf`**

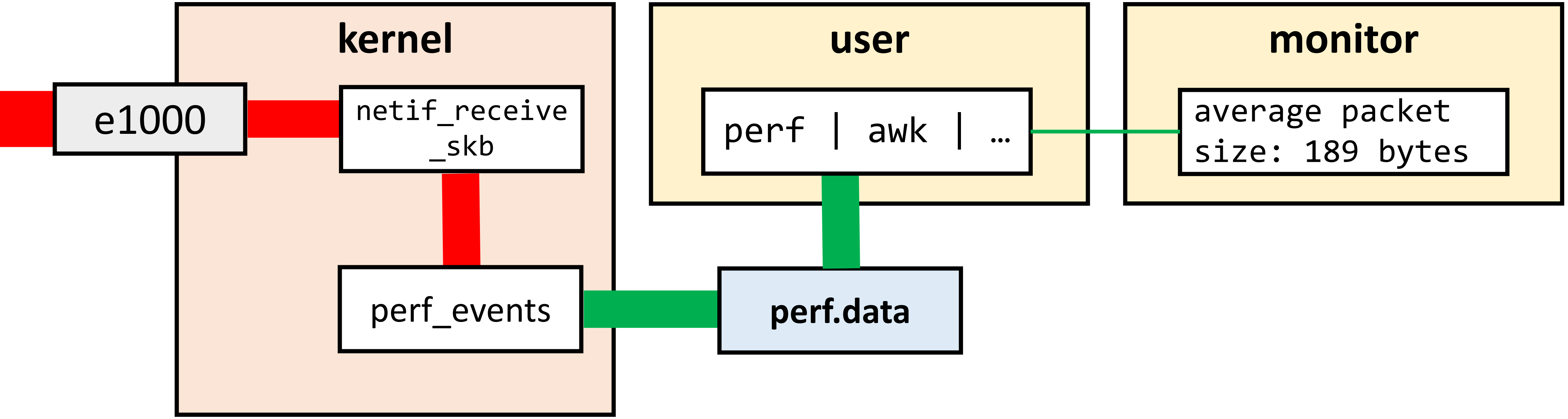
- There is at least one talk at every Java conference about using `perf` for CPU profiling and flame graphs
  - JPoint 2017: Пангин и Цесько, JVM-профайлер с чувством такта
- For example:

```
# perf record -g -F 97 -- java …
# perf script | ./stackcollapse-perf.pl | ./flamegraph.pl --color java > java.svg
```



# What’s Wrong With `perf`?

- `perf` relies on pushing a *lot of data* to user space, through *files*, for *analysis*
  - Downloading a file at ~1Gb/s produces ~89K `netif_receive_skb` events/s (19MB/s including stacks)



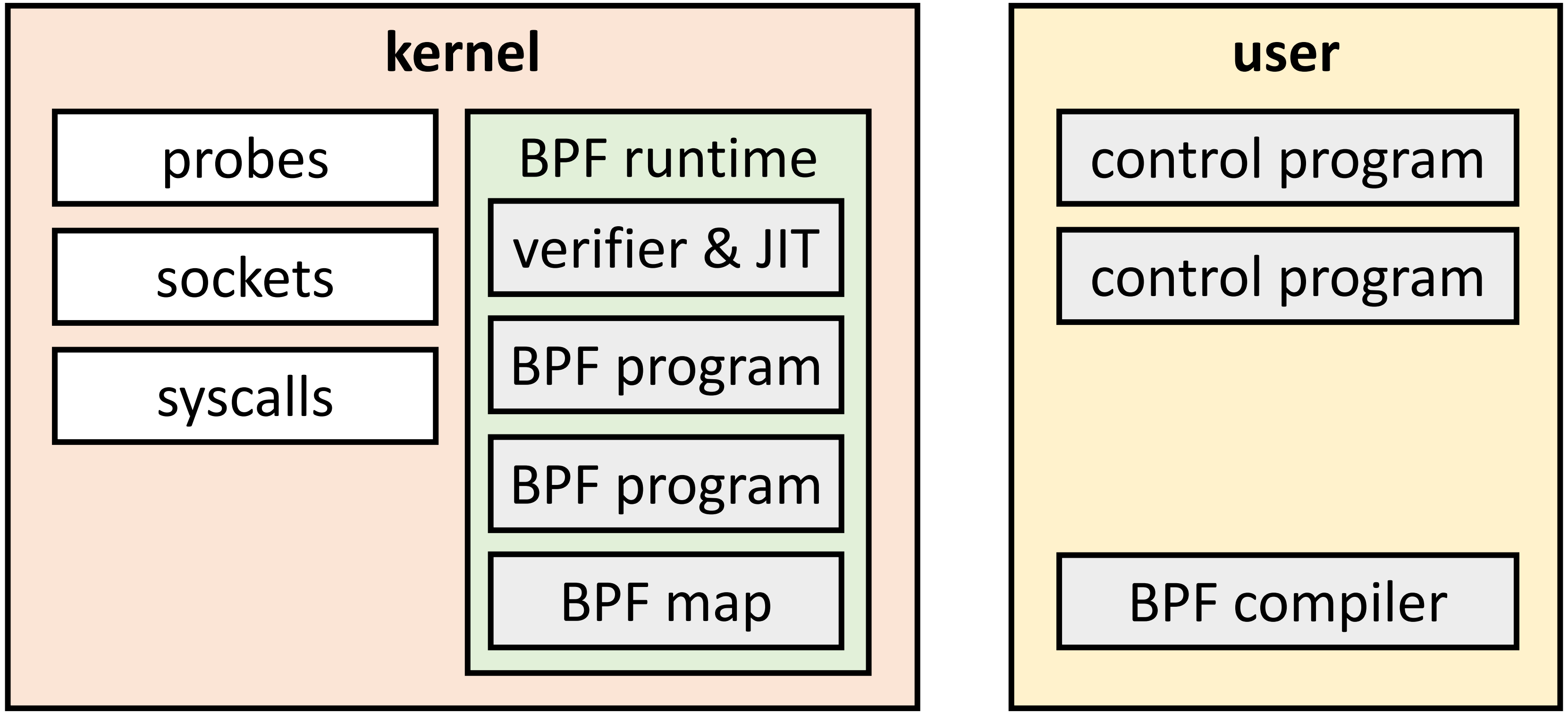
# BPF: 1990

- Invented by McCanne and Jacobson at Berkeley, 1990-1992: instruction set, representation, implementation of packet filters

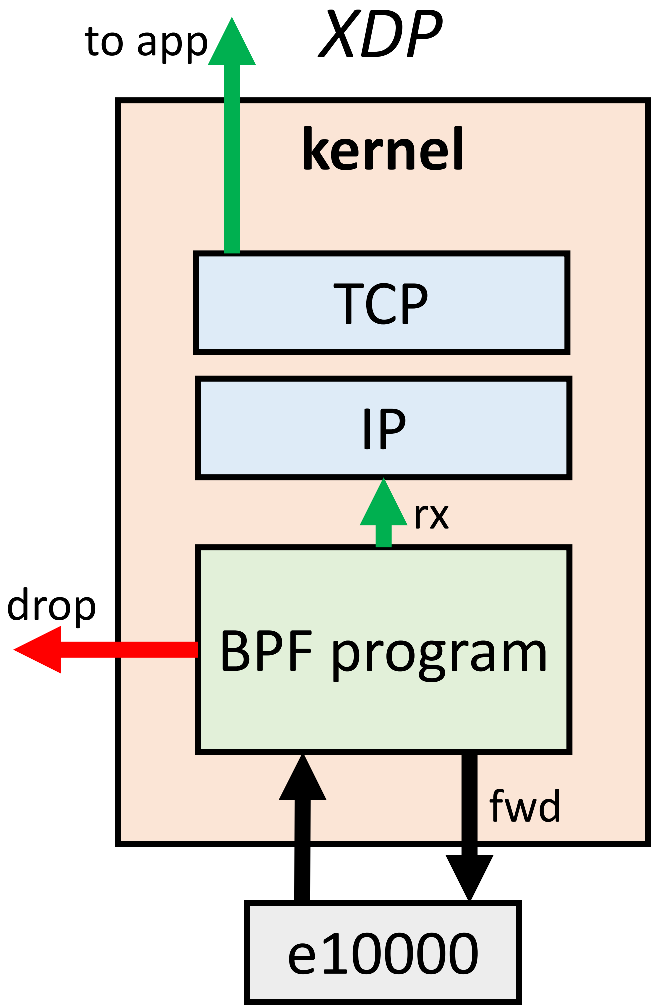
```
$ tcpdump -d 'ip and dst 186.173.190.239'
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 5
(002) ld       [30]
(003) jeq      #0xbaadbeef      jt 4    jf 5
(004) ret      #262144
(005) ret      #0
```

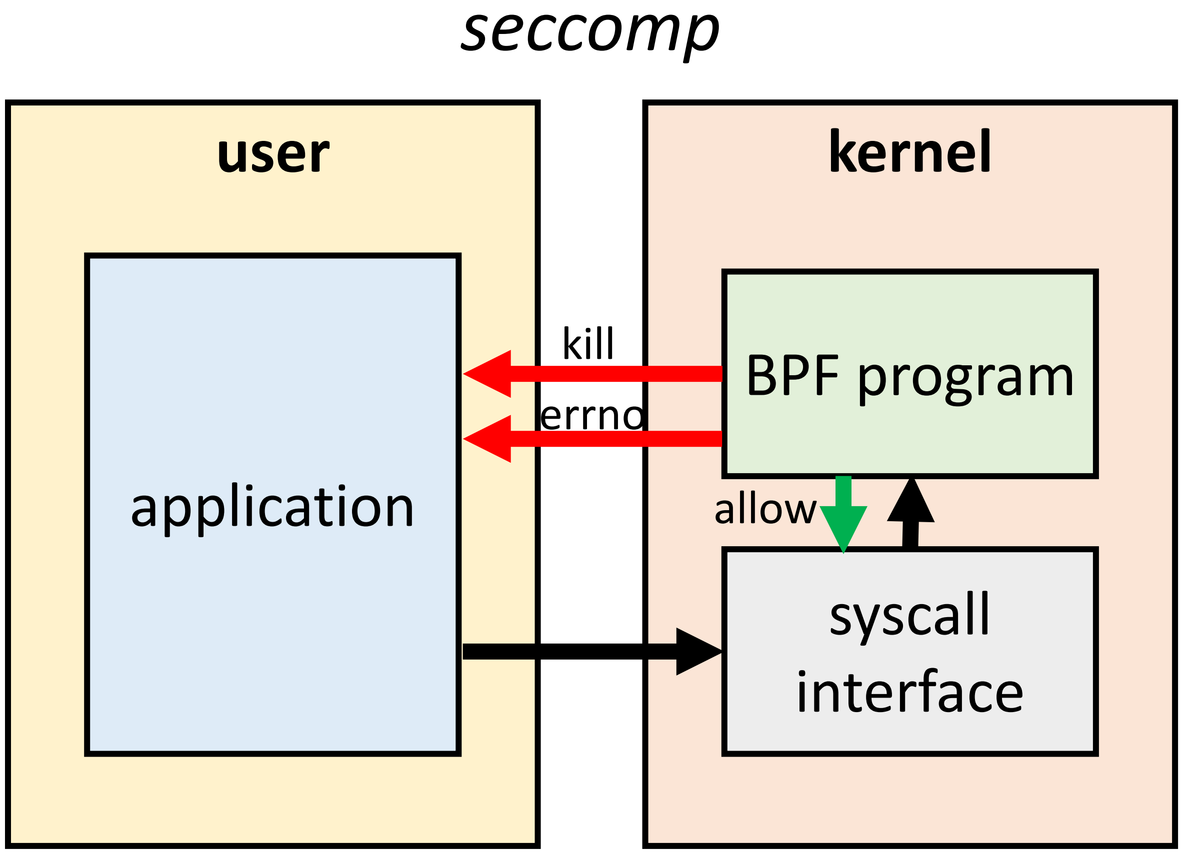
# BPF: Today

- Supports a wide spectrum of usages
- Has a JIT for maximum efficiency

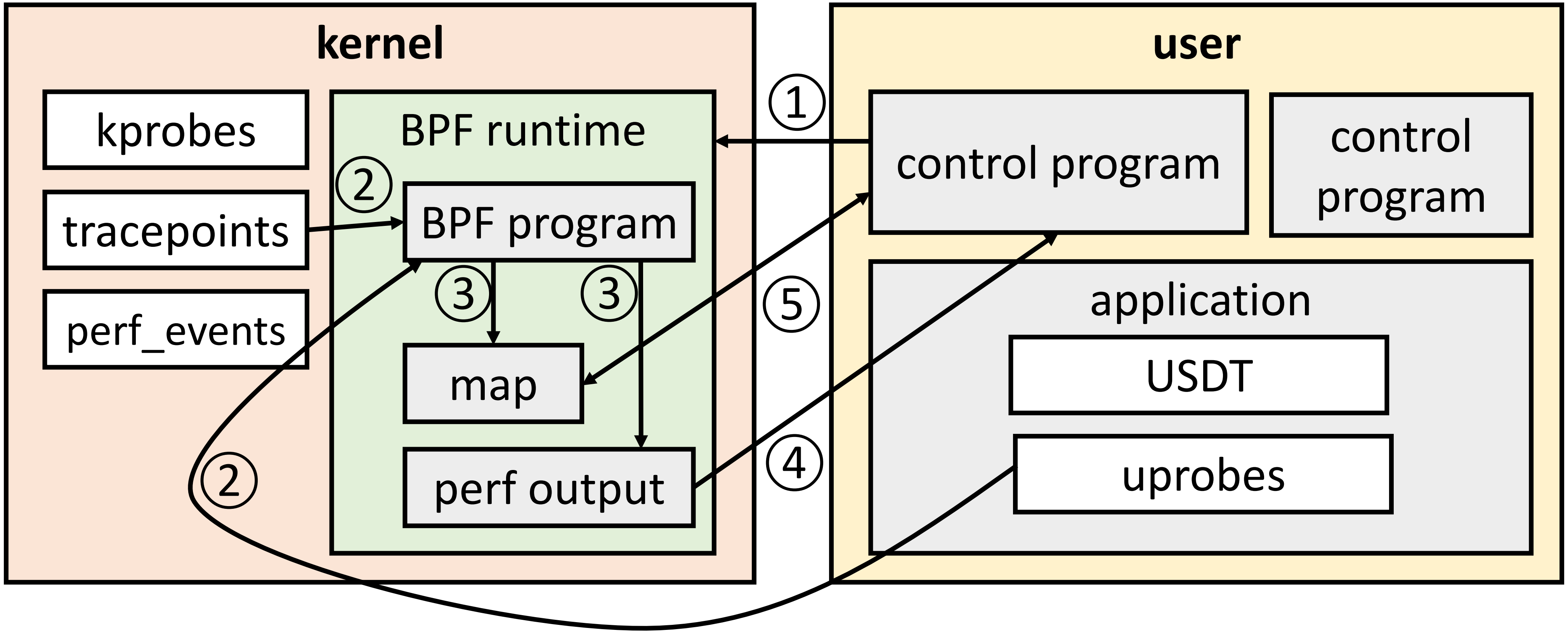


# BPF Scenarios





# BPF Tracing



① installs BPF program and attaches to events
② events invoke the BPF program
③ BPF program updates a map or pushes a new event to a buffer shared with user-space

④ user-space program is invoked with data from the shared buffer
⑤ user-space program reads statistics from the map and clears it if necessary

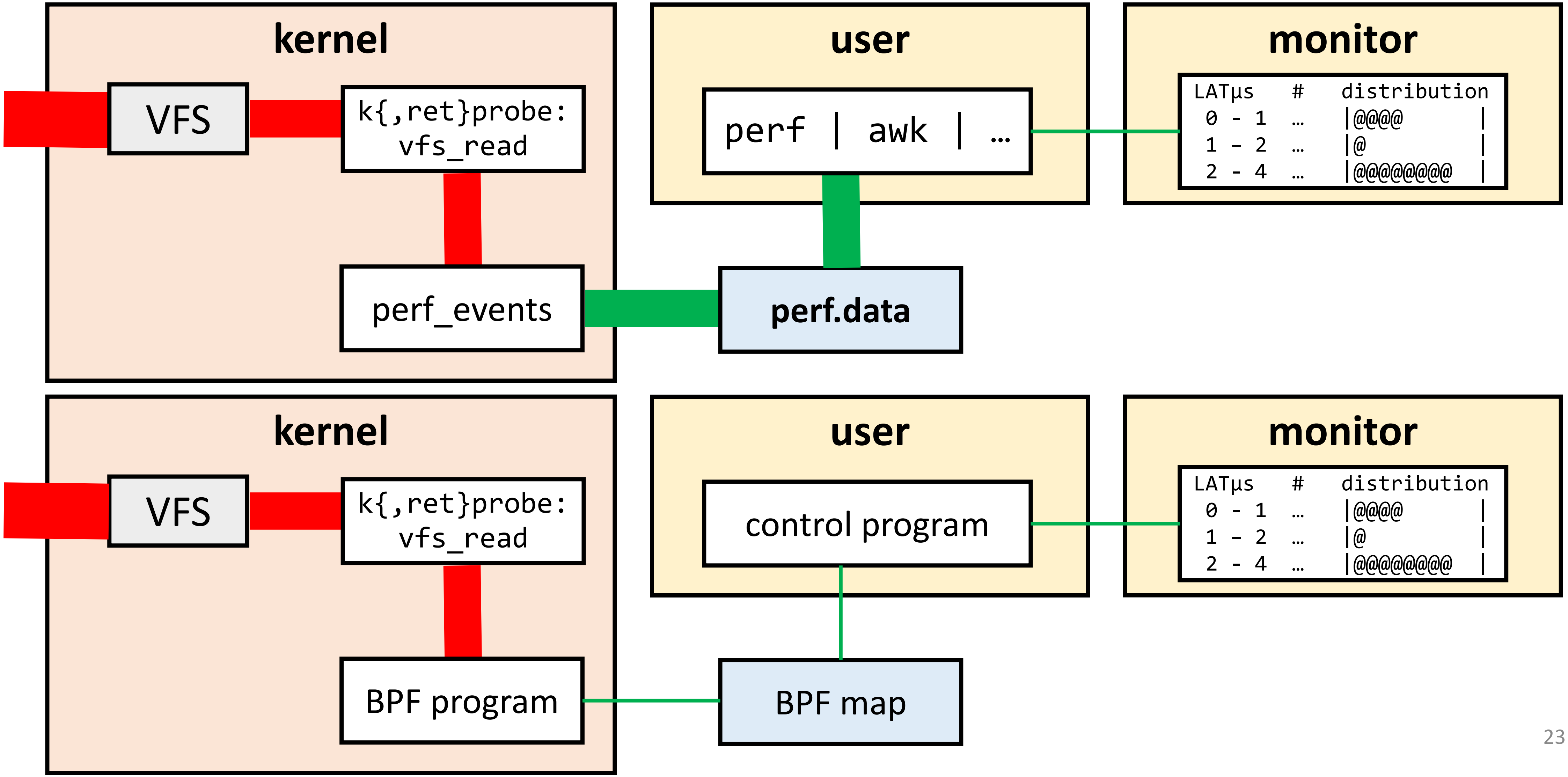
# BPF Tracing Features in The Linux Kernel

| Version | Feature | Scenarios |
|---|---|---|
| 4.1 | kprobes/uprobes attach | Dynamic tracing with BPF becomes possible |
| 4.1 | `bpf_trace_printk` | BPF programs can print output to ftrace pipe |
| 4.3 | perf_events output | Efficient tracing of large amounts of data for analysis in user-space |
| 4.6 | Stack traces | Efficient aggregation of call stacks for profiling or tracing |
| 4.7 | Tracepoints support | API stability for tracing programs |
| 4.9 | perf_events attach | Low-overhead profiling and PMU sampling |

24
16.04

25
16.10

# The Old Way And The New Way



**kernel**

VFS

`k{,ret}probe: vfs_read`

perf_events

**user**

`perf | awk | …`

**perf.data**

**monitor**

```
LATµs   #   distribution
 0 - 1  …   |@@@@       |
 1 - 2  …   |@          |
 2 - 4  …   |@@@@@@@@   |
```

**kernel**

VFS

`k{,ret}probe: vfs_read`

BPF program

**user**

control program

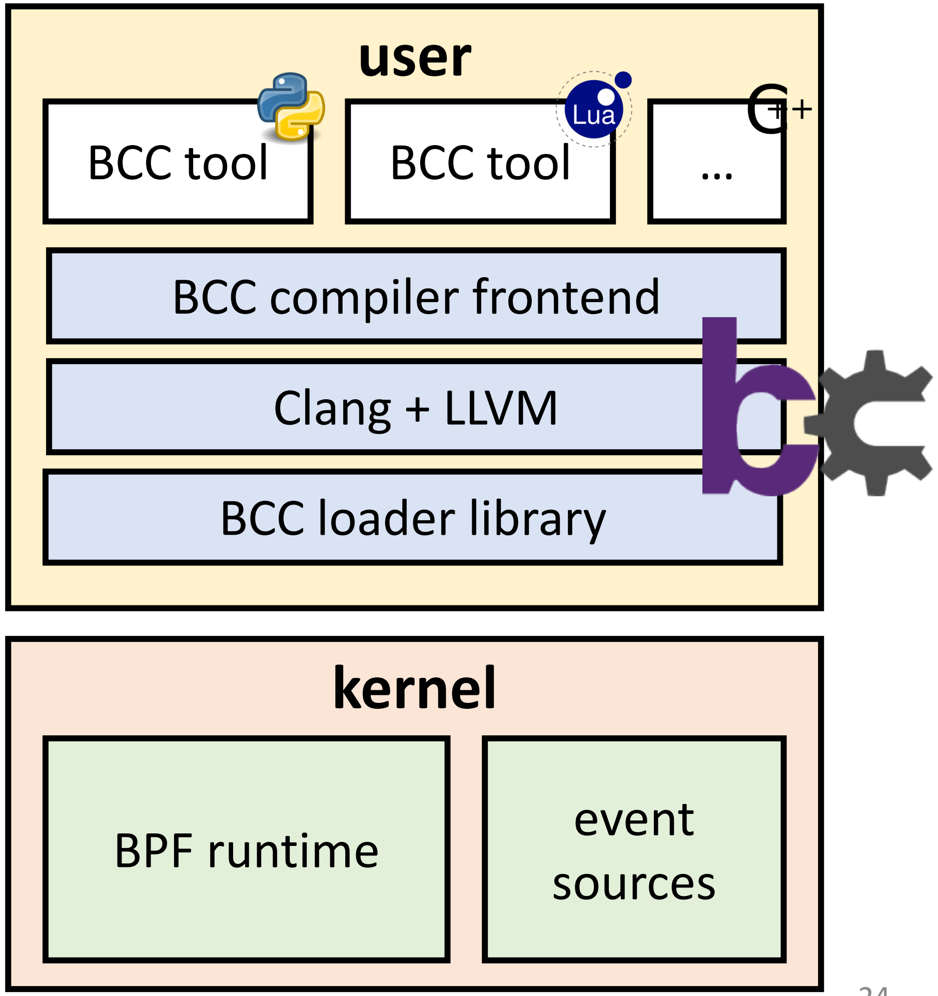
BPF map

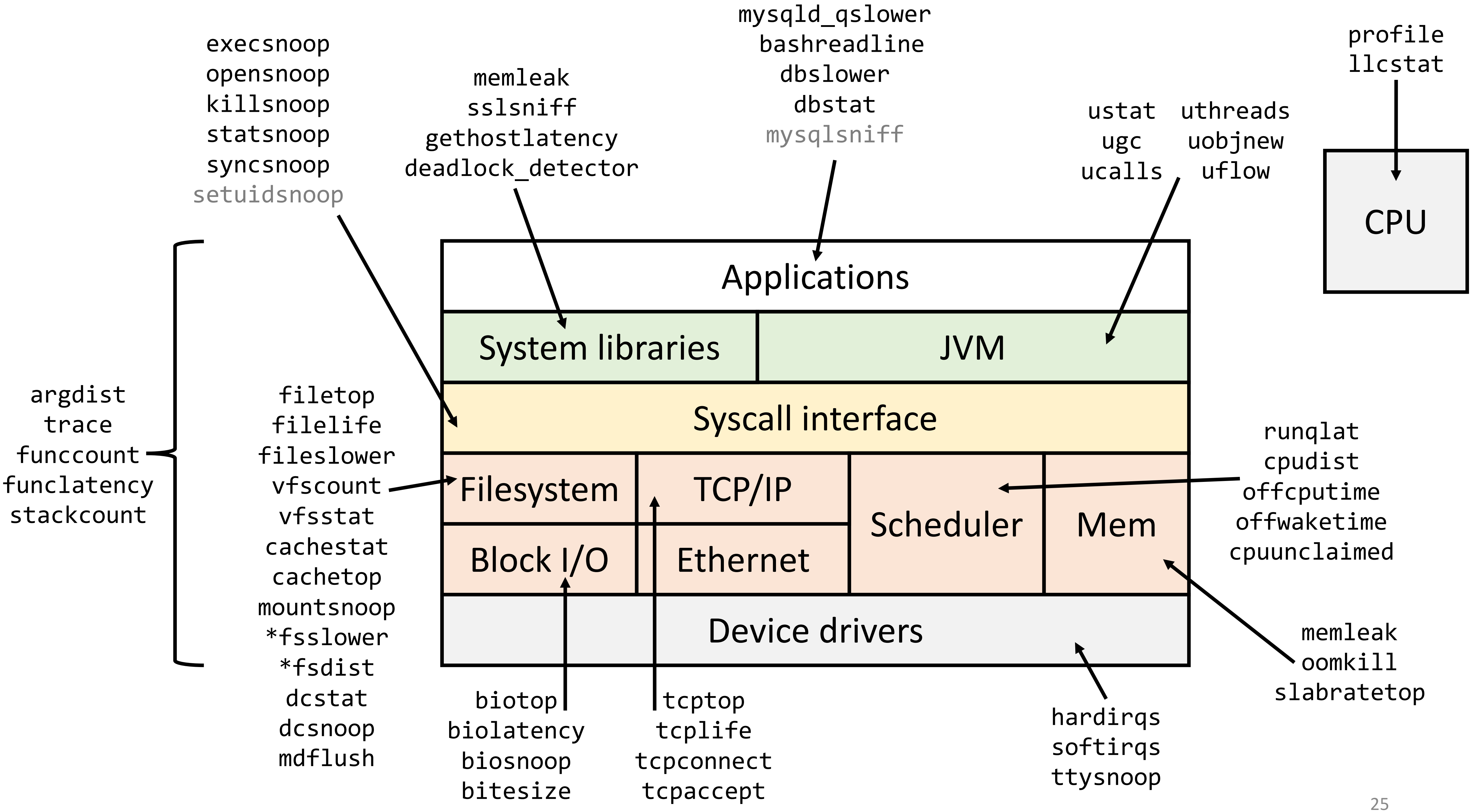
**monitor**

```
LATµs   #   distribution
 0 - 1  …   |@@@@       |
 1 - 2  …   |@          |
 2 - 4  …   |@@@@@@@@   |
```

# The BCC BPF Front-End

- https://github.com/iovisor/bcc
- BPF Compiler Collection (BCC) is a BPF frontend library and a massive collection of performance tools
  - Contributors from Facebook, PLUMgrid, Netflix, Sela
- Helps build BPF-based tools in high-level languages
  - Python, Lua, C++





execsnoop
opensnoop
killsnoop
statsnoop
syncsnoop
setuidsnoop

memleak
sslsniff
gethostlatency
deadlock_detector

mysqld_qslower
bashreadline
dbslower
dbstat
mysqlsniff

ustat
ugc
ucalls

uthreads
uobjnew
uflow

profile
llcstat

CPU

Applications

System libraries

JVM

Syscall interface

Filesystem

TCP/IP

Scheduler

Mem

Block I/O

Ethernet

Device drivers

argdist
trace
funccount
funclatency
stackcount

filetop
filelife
fileslower
vfscount
vfsstat
cachestat
cachetop
mountsnoop
*fsslower
*fsdist
dcstat
dcsnoop
mdflush

runqlat
cpudist
offcputime
offwaketime
cpuunclaimed

memleak
oomkill
slabratetop

biotop
biolatency
biosnoop
bitesize

tcptop
tcplife
tcpconnect
tcpaccept

hardirqs
softirqs
ttysnoop

# BCC JVM on Linux Performance Checklist

1. `ustat`
2. `ugc`
3. `execsnoop`
4. `opensnoop`
5. `ext4slower`
   (or `btrfs*`, `xfs*`, `zfs*`)
6. `biolatency`
7. `biosnoop`
8. `cachestat`
9. `tcpconnect`
10. `tcpaccept`
11. `tcptop`
12. `gethostlatency`
13. `uthreads`
14. `cpudist`
15. `runqlat`
16. `profile`

# Demo: CPU Investigation 👀

- Objective:
  Identify on-CPU hot methods in a running application

```
$ top
  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
25491 vagrant   20   0 2260396  29372  15876 S 99.3  2.9   0:15.93 java
# uthreads -l java `pidof java`
Tracing thread events in process 25582 (language: java)... Ctrl-C to quit.
TIME     ID               TYPE     DESCRIPTION
1.061    25594            pthread  [unknown]
1.062    R=8/N=0          start    Thread-0
1.068    25595            pthread  [unknown]
1.069    R=9/N=0          start    Thread-1
…
# profile -U -p `pidof java` -F 97 5
…
    Primes::isPrime
    Primes::primesThread
    Primes::access$000
    Primes$1::run
    java.lang.Thread::run
    call_stub
…
    start_thread
    -                java (25582)
        43
```

`uthreads` (from BCC) traces thread creation and destruction events

`profile` (from BCC) collects hot CPU call stacks and aggregates them (requires **/tmp/perf-*PID*.map**)

```
# profile -f -F 97 5 > java.stacks
$ cat java.stacks | flamegraph.pl --color java > javacpu.svg
```

`profile -f` outputs folded stacks in a format suitable for flame graph generation

```
$ grep -A3 -B1 synchronized Computey.java
                        if (isPrime(i)) {
                                synchronized (primesLock) {
                                        primes.add(i);
                                }
                        }
# argdist -p `pidof java` -i 5 -C 'u:libjvm.so:monitor__contended__enter()'
[11:22:06]
u:libjvm.so:monitor__contended__enter()
        COUNT      EVENT
[11:22:11]
u:libjvm.so:monitor__contended__enter()
        COUNT      EVENT
        3          total calls
[11:22:16]
u:libjvm.so:monitor__contended__enter()
        COUNT      EVENT
[11:22:21]
u:libjvm.so:monitor__contended__enter()
        COUNT      EVENT
[11:22:26]
u:libjvm.so:monitor__contended__enter()
        COUNT      EVENT
        9          total calls
```

`argdist` (from BCC) generates frequency counts or histograms of interesting events

# Demo: Slow MySQL Queries 👀

- Objective:
  Determine why a Java + MySQL application occasionally produces slow results, and where the slow queries are coming from

```
# dbstat mysql
       query latency (ms)  : count     distribution
         0 -> 1          : 3027      |****************************************|
         2 -> 3          : 5         |                                        |
         4 -> 7          : 1         |                                        |
         8 -> 15         : 0         |                                        |
        16 -> 31         : 0         |                                        |
        32 -> 63         : 0         |                                        |
        64 -> 127        : 0         |                                        |
       128 -> 255        : 0         |                                        |
       256 -> 511        : 0         |                                        |
       512 -> 1023       : 0         |                                        |
      1024 -> 2047       : 8         |                                        |
```

dbstat (from BCC) generates a histogram of database query latencies for MySQL and PostgreSQL

```
# dbstat mysql -m 5
Tracing database queries slower than 5ms for PID 25776... Ctrl+C to quit.
       query latency (ms)  : count     distribution

…

        16 -> 31         : 0         |                                        |
        32 -> 63         : 0         |                                        |
        64 -> 127        : 0         |                                        |
       128 -> 255        : 0         |                                        |
       256 -> 511        : 0         |                                        |
       512 -> 1023       : 0         |                                        |
      1024 -> 2047       : 6         |****************************************|
```

```
$ top
  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
25776 mysql     20   0 1201036 101120  15144 S  2.0 10.0   0:03.80 mysqld
26036 vagrant   20   0 2261580  59580  16344 S  1.3  5.9   0:07.80 java
# dbslower mysql -m 500
Tracing database queries for PID 25776 slower than 500 ms...
TIME(s)        PID          MS QUERY
0.000000       25776  2001.779 call getproduct(97)
2.123951       25776  2002.256 call getproduct(97)
4.259418       25776  2002.404 call getproduct(97)
6.387346       25776  2002.110 call getproduct(97)
```

`dbslower` (from BCC) displays MySQL/PostgreSQL queries slower than the specified threshold

```
# trace -p `pidof mysqld` 'u:/usr/local/mysql/bin/mysqld:query__exec__start "query=%s", arg1'
PID    TID    COMM         FUNC             -
25776  26047  mysqld       query__exec__start query=call getproduct(95)
25776  26047  mysqld       query__exec__start query=select * from products where id = 95
25776  26047  mysqld       query__exec__start query=select * from users where id = 48
25776  26047  mysqld       query__exec__start query=select id from products where userid = 48
25776  26047  mysqld       query__exec__start query=call getproduct(96)
25776  26047  mysqld       query__exec__start query=select * from products where id = 96
25776  26047  mysqld       query__exec__start query=call getproduct(97)
25776  26047  mysqld       query__exec__start query=do sleep(2)
…
```

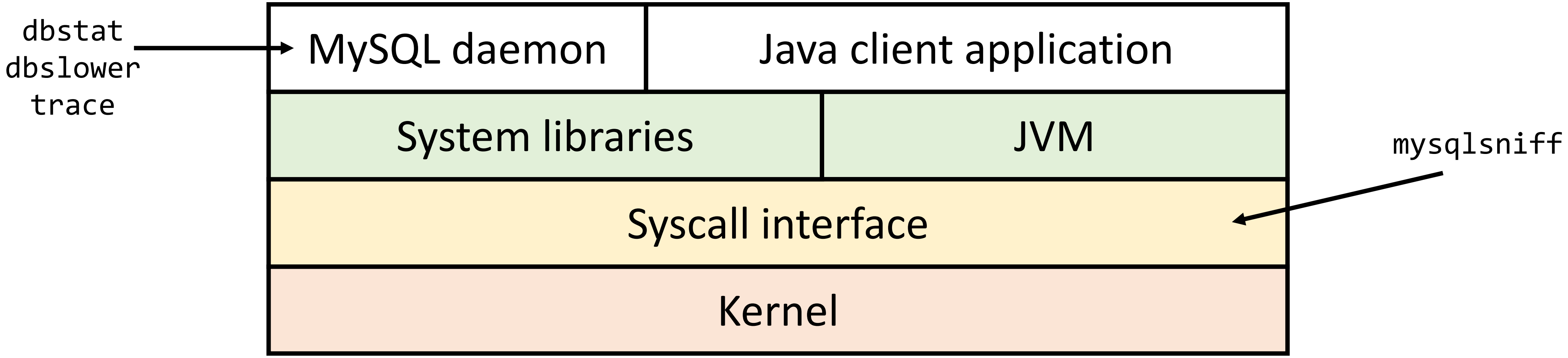
by attaching `trace` to the `query__exec__start` probe, we see the internal queries executed by the `getproduct` sproc

```
# ./mysqlsniff.py -p `pidof java` -f "call getproduct(97)" -S
Sniffing process 26036, Ctrl+C to quit.
call getproduct(97)
        __libc_send+0x0
        Java_java_net_SocketOutputStream_socketWrite0+0x102
        java.net.SocketOutputStream::socketWrite0+0xda
        java.net.SocketOutputStream::socketWrite+0x84
        java.net.SocketOutputStream::write+0x34
        java.io.BufferedOutputStream::flushBuffer+0x5c
        java.io.BufferedOutputStream::flush+0x78
        com.mysql.jdbc.MysqlIO::send+0x2d0
        com.mysql.jdbc.MysqlIO::sendCommand+0x188
        com.mysql.jdbc.MysqlIO::sqlQueryDirect+0x8f8
        com.mysql.jdbc.ConnectionImpl::execSQL+0x324
        com.mysql.jdbc.ConnectionImpl::execSQL+0x74
        com.mysql.jdbc.StatementImpl::executeQuery+0x4ec
        Product::load+0x288
        User::loadProducts+0x33c
        Databasey::main+0x16b
        call_stub+0x88
…
        start_thread+0xca
```

`mysqlsniff` (demo tool) analyzes client network traffic to identify MySQL queries and display call stack

# Demo Summary: Slow MySQL Queries 👀

- Objective:
  Determine why a Java + MySQL application occasionally produces slow results, and where the slow queries are coming from



# Demo: Lots of GC 👀

- Objective:
  Understand the load generated by a Java application, identify that it has to do with GC, and figure out where the garbage is coming from

```
$ top
  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
26176 vagrant   20   0 2256284  45276  15884 S 99.9  4.5   0:07.10 java
# ustat -C
15:02:09 loadavg: 0.24 0.05 0.03 3/202 28698
PID    CMDLINE              METHOD/s   GC/s   OBJNEW/s   CLOAD/s  EXC/s  THR/s
28689  java -XX:+PreserveFr 0          888    0          0        0      0

15:02:10 loadavg: 0.30 0.07 0.04 4/202 28698
PID    CMDLINE              METHOD/s   GC/s   OBJNEW/s   CLOAD/s  EXC/s  THR/s
28689  java -XX:+PreserveFr 0          898    0          0        0      0
# profile -p `pidof java` 5
…
    __memset_erms
    [unknown]
    _new_array_Java
    ResponseBuilder::addLine
    Allocy::main
    call_stub
…
    start_thread
    -                java (26176)
        31
```

ustat (from BCC) is a top-like extension for Java events (and other languages)

```
# uobjnew java `pidof java` 5
Tracing allocations in process 26259 (language: java)... Ctrl-C to quit.
TYPE                           # ALLOCS      # BYTES
java/lang/String                  12588            0
[C                                12588            0

TYPE                           # ALLOCS      # BYTES
java/lang/String                  11680            0
[C                                11680            0
# stackcount -i 5 -p `pidof java` "u:…/libjvm.so:object__alloc"
…
  SharedRuntime::dtrace_object_alloc(oopDesc*, int)
  TypeArrayKlass::allocate_common(int, bool, Thread*)
  OptoRuntime::new_array_C(Klass*, int, JavaThread*)
  _new_array_Java
  ResponseBuilder::addLine
  Allocy::main
  call_stub
…
  JavaMain
  start_thread
    870
```

🤔[C😌

uobjnew (from BCC) traces object allocations by using an expensive probe that requires `-XX:+ExtendedDTraceProbes`

`stackcount` (from BCC) attaches to that probe and summarizes Java call stacks leading up to it (could also do a flame graph)

# Demo Summary : Lots of GC 👀

- Objective:
  Understand the load generated by a Java application, identify that it has to do with GC, and figure out where the garbage is coming from
- Commands used:

```
$top
#ustat
#profile -U -p `pidof java` 5
#uobjnew java `pidof java` 5
#stackcount -i 5 -p `pidof java`
            "u:…/libjvm.so:object__alloc"
```

# Demo: Failed Initialization 👀

- Objective:
  Figure out why an application fails to initialize and keeps printing weird messages

```
$ java … Servery
[*] Server started, initializing.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
[*] Opening config file.
```

```
# opensnoop -x
PID    COMM               FD ERR PATH
26456  java               -1   2 /etc/acme-svr.config
26456  java               -1   2 /etc/acme-svr.config
26456  java               -1   2 /etc/acme-svr.config
26456  java               -1   2 /etc/acme-svr.config
# trace -p `pidof java` -U 'r::SyS_open (retval==-2) "failed open"'
PID    TID    COMM         FUNC             -
26456  26466  java         SyS_open         failed open
        __open64+0x2d [libpthread-2.24.so]
        fileOpen+0x7a [libjava.so]
        java.io.FileInputStream::open0+0xbc [perf-16028.map]
        Initializer::openConfigFile->
          java.util.Scanner::<init>->
          java.io.FileInputStream::<init>->
          java.io.FileInputStream::open+0x0 [perf-16028.map]
        Initializer$1::run->Initializer::access$100+0x0 [perf-16028.map]
        java.lang.Thread::run+0x13d [perf-16028.map]
        call_stub+0x88 [perf-16028.map]
…
        java_start(Thread*)+0xf2 [libjvm.so]
        start_thread+0xca [libpthread-2.24.so]
```

opensnoop (from BCC) traces file open events and displays the path and errors, if any

we trace the open syscall and print the user stack if the syscall failed

# Demo: Slow HTTP Requests 👀

- Objective:
  Figure out why an HTTP client application occasionally makes very slow requests

```
$ java -cp bin Clienty good
Crawl complete, elapsed: 2241 milliseconds.
Crawl complete, elapsed: 1614 milliseconds.
Crawl complete, elapsed: 1442 milliseconds.
Crawl complete, elapsed: 1467 milliseconds.
Crawl complete, elapsed: 1601 milliseconds.
Crawl complete, elapsed: 1599 milliseconds.
…
$ java -cp bin Clienty bad
Crawl complete, elapsed: 6442 milliseconds.
Crawl complete, elapsed: 6060 milliseconds.
Crawl complete, elapsed: 6044 milliseconds.
Crawl complete, elapsed: 6092 milliseconds.
Crawl complete, elapsed: 6031 milliseconds.
Crawl complete, elapsed: 6023 milliseconds.
…
$ grep -A4 bad Clienty.java
                if (args[0].equals("bad")) {
                        urls.add("https://i-dont-exist-at-all-20170126.com");
                } else {
                        urls.add("https://facebook.com");
                }
```

```
# gethostlatency
TIME      PID    COMM                  LATms HOST
15:18:11  29003  java                6021.00 i-dont-exist-at-all-20170126.com
15:18:17  29003  java                6030.00 i-dont-exist-at-all-20170126.com
15:18:23  29003  java                6029.00 i-dont-exist-at-all-20170126.com
# trace -T -p `pidof java` 'c:getaddrinfo "resolving: %s", arg1' \
                           'r:c:getaddrinfo "done resolving: %d", retval'
TIME     PID    TID    COMM         FUNC              -
16:21:55 15611  15612  java         getaddrinfo       resolving: i-dont-exist-…com
16:22:01 15611  15612  java         getaddrinfo       done resolving: -2
16:22:01 15611  15612  java         getaddrinfo       resolving: i-dont-exist-…com
16:22:07 15611  15612  java         getaddrinfo       done resolving: -2
16:22:07 15611  15612  java         getaddrinfo       resolving: i-dont-exist-….com
16:22:13 15611  15612  java         getaddrinfo       done resolving: -2
```

```
#define ENOENT 2
```

gethostlatency (from BCC) traces DNS resolution latency

```
$ dig +multiline +answer any i-dont-exist-at-all-20170126.com
…
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 20042
…
;; AUTHORITY SECTION:
com.                    0   IN SOA a.gtld-servers.net. nstld.verisign-grs.com. (
                                1486992927 ; serial
                                1800       ; refresh (30 minutes)
                                900        ; retry (15 minutes)
                                604800     ; expire (1 week)
                                86400      ; minimum (1 day)
                                )

;; Query time: 3020 msec
…
$ dig +multiline +answer any i-dont-exist-at-all-20170126.com
…
;; AUTHORITY SECTION:
com.                    0   IN SOA a.gtld-servers.net. nstld.verisign-grs.com. (
…
```

dig helps debug DNS resolution issues

```
$ dig +multiline +answer any i-dont-exist-at-all-20170126.com
…
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 20042
…
;; AUTHORITY SECTION:
com.                    828 IN SOA a.gtld-servers.net. nstld.verisign-grs.com. (
                                1486992927 ; serial
                                1800       ; refresh (30 minutes)
                                900        ; retry (15 minutes)
                                604800     ; expire (1 week)
                                86400      ; minimum (1 day)
                                )

;; Query time: 3020 msec
…
$ less Clienty.java
…
        static {
                Security.setProperty(
                                "networkaddress.cache.negative.ttl", "0");
        }
…
```

`networkaddress.cache.negative.ttl`
controls the cache lifetime of failed DNS requests

# Summary

- We have seen:

✓ Which production-ready tracing tools can be used with JVM apps

✓ How BPF changes the picture of Linux tracing

✓ To apply a performance checklist for JVM apps using BPF tools

✓ To conduct ad-hoc investigations with one-liners and custom tools

# References

- BPF
  - https://github.com/torvalds/linux/tree/master/samples/bpf
  - https://www.kernel.org/doc/Documentation/networking/filter.txt
  - https://github.com/iovisor/bpf-docs
- BCC tutorials (by Brendan Gregg)
  - https://github.com/iovisor/bcc/blob/master/docs/tutorial.md
  - https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md
  - https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md
- JVM USDT probes
  - http://blog.sashag.net/2016/12/23/usdtbpf-tracing-tools-java-python-ruby-node-mysql-postgresql/
  - https://docs.oracle.com/javase/8/docs/technotes/guides/vm/dtrace.html
  - https://sourceware.org/systemtap/wiki/AddingUserSpaceProbingToApps

# Thank You!

Slides: https://s.sashag.net/jpoint17

Demos & labs: https://github.com/goldshtn/linux-tracing-workshop

Sasha Goldshtein

CTO, Sela Group

goldshtn

goldshtn