Strategic Sponsors

KMD
KMDPOLAND.PL

7N

Gold Sponsors

sonova
HEAR THE WORLD

VOLVO
Volvo Group IT

Relativity

ABB

Silver Sponsors

JCommerce

RAVENDB

it KONTRAKT

DevExpress

Millennium bank

IT outsourcing nbc

labs software specialists

ASPOSE File Format APIs

Demant Technology Centre

LUXOFT

Sasha Goldshtein, CTO, Sela Group
Microsoft Regional Director, C# MVP
@goldshtn

Debugging and Profiling .NET Core Apps on Linux

The Plan

This is a talk on debugging and profiling .NET Core apps on Linux—yes, it's pretty crazy that we got this far!

You'll learn:

- ❑ To profile CPU activity in .NET Core apps
- ❑ To visualize stack traces (e.g. of CPU samples) using flame graphs
- ❑ To use Linux tracing tools with .NET Core processes
- ❑ To capture .NET Core runtime events using LTTng
- ❑ To generate and analyze core dumps of .NET Core apps

Disclaimer

Some of this stuff is changing quite rapidly

The tools described here sort of work for .NET Core 2.0, but your mileage may vary

Some of this relies on scripts I hacked together, and will hopefully be officially supported in the future

Tools And Operating Systems Supported

	Linux	Windows	macOS
CPU sampling	perf, BCC	ETW	Instruments, dtrace
Dynamic tracing	perf, SystemTap, BCC	⊖	dtrace
Static tracing	LTTng	ETW	⊖
Dump generation	core_pattern, gcore	Procdump, WER	kern.corefile, gcore
Dump analysis	lldb	Visual Studio, WinDbg	lldb
	This talk		

⚠ Mind The Overhead

.NET DEVELOPER DAYS

Any observation can change the state of the system, but some observations are worse than others

Diagnostic tools have overhead

- Check the docs
- Try on a test system first
- Measure degradation introduced by the tool

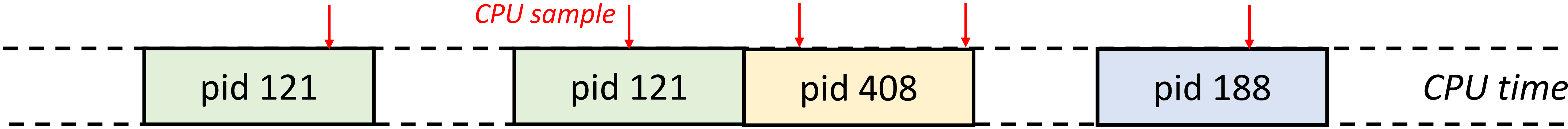
OVERHEAD

> This traces various kernel page cache functions and maintains in-kernel counts, which are asynchronously copied to user-space. While the rate of operations can be very high (>1G/sec) we can have up to 34% overhead, this is still a relatively efficient way to trace these events, and so the overhead is expected to be small for normal workloads. Measure in a test environment.
>
> —*`man cachestat`* *(from BCC)*

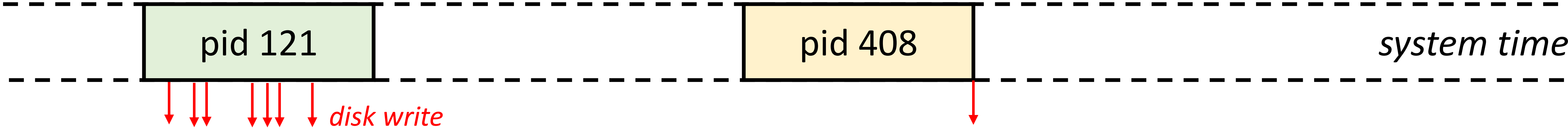
Sampling vs. Tracing

Sampling works by getting a snapshot or a call stack every N occurrences of an interesting event

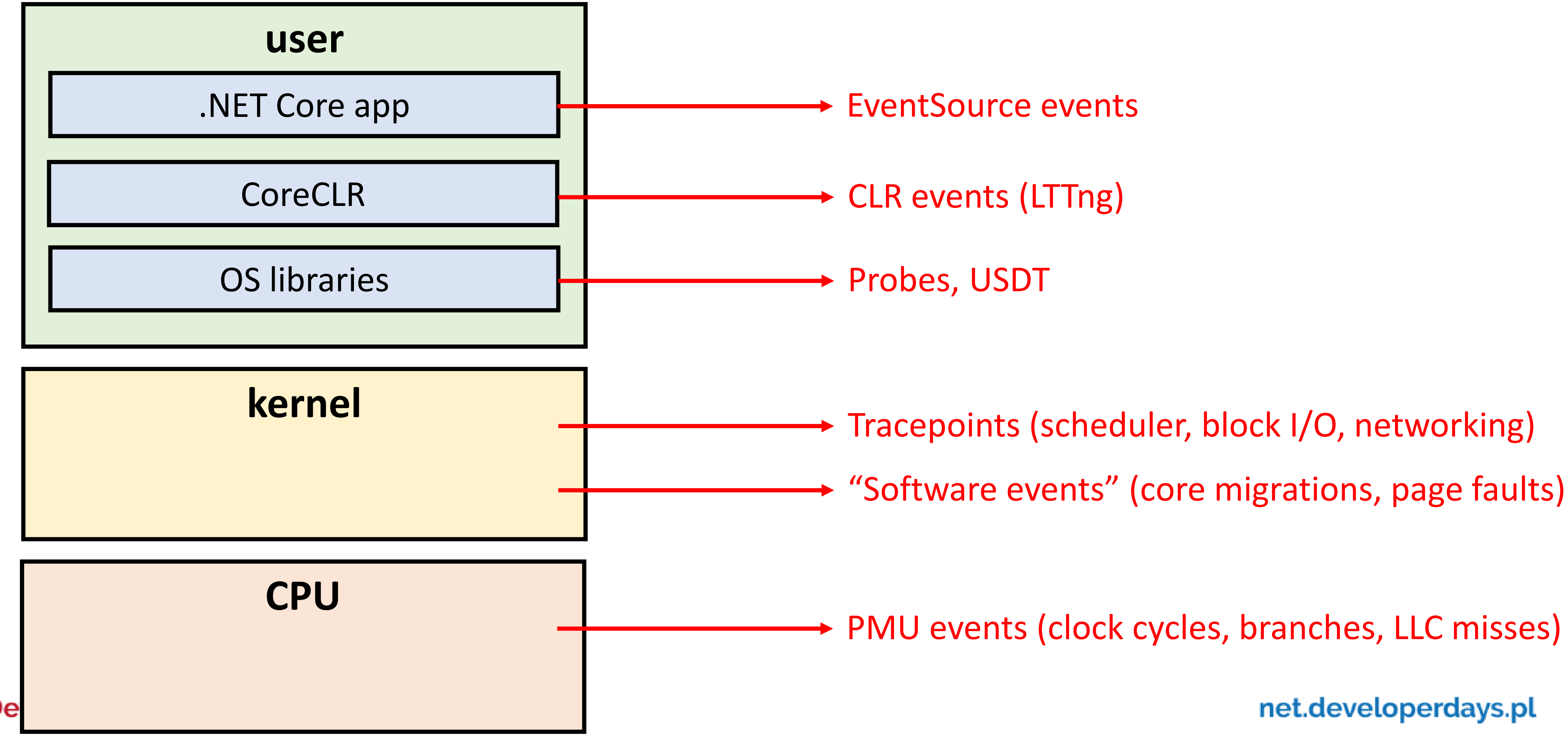
For most events, implemented in the PMU using overflow counters and interrupts



Tracing works by getting a message or a call stack at every occurrence of an interesting event



.NET Core on Linux Tracing Architecture



The Official Story: **perfcollect** and PerfView

1. Download perfcollect
2. Install prerequisites: `./perfcollect install`
3. Run collection: `./perfcollect collect mytrace`
4. Copy the mytrace.zip file to a Windows machine 😳
5. Download PerfView 😁
6. Open the trace in PerfView 😱

perf

perf is a Linux multi-tool for performance investigations

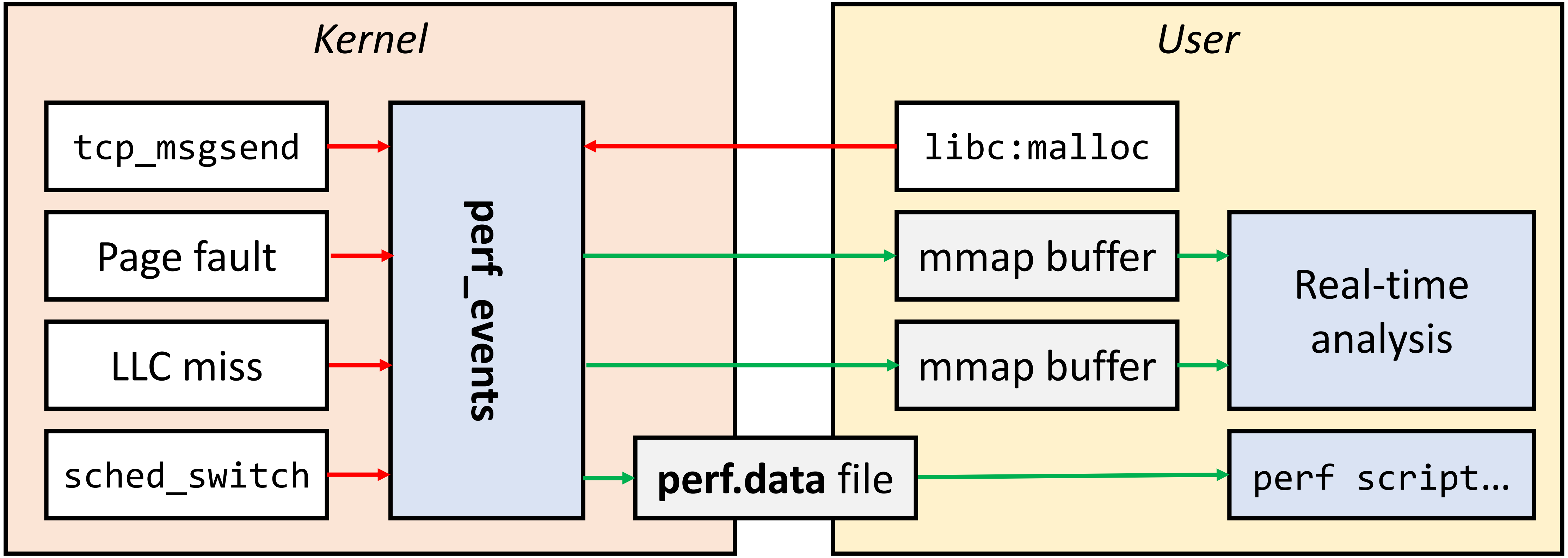
Capable of both tracing and sampling

Developed in the kernel tree, must match running kernel's version

Debian-based: `apt install linux-tools-common`

RedHat-based: `yum install perf`

perf_events Architecture



Five Things That Will Happen To You If You Don't Have Symbolic Debug Information

.NET DEVELOPER DAYS

```
 Children      Self  Command  Shared Object              Symbol
 ........  ........  .......  .........................  ...............................................

   70.03%     0.00%  Buggy    perf-4644.map              [.] 0x00007f3ca95e0c55
           |
           ---0x7f3ca95e0c55
              |
              |--69.19%--0x7f3ca95f007b
              |          |
              |          |--41.46%--0x4af45a
              |          |          |
              |          |           --40.90%--0x4aec4d
              |          |                     |
              |          |                      --40.62%--0x4d3f70
              |          |                                __memmove_avx_unaligned_erms
              |          |                                |
              |          |                                 --12.32%--page_fault
              |          |                                           do_page_fault
              |          |                                           |
              |          |                                            --12.04%--__do_page_fault
              |          |                                                      |
              |          |                                                      |--5.04%--handle_mm_fault
```

The stuff we care about

Getting Debug Information

	Type	Debug information source
`SyS_write`	Kernel	`/proc/kallsyms`
`__write`	Native	Debuginfo package
`System.IO.SyncText…`	Managed (AOT)	Crossgen*
`System.Console.WriteLine`	Managed (AOT)	Crossgen*
`MyApp.Program.Foo`	Managed (JIT)	`/tmp/perf-$PID.map`
`MyApp.Program.Main`	Managed (JIT)	`/tmp/perf-$PID.map`
`ExecuteAssembly`	Native (CLR)	Debuginfo package or source build
`CorExeMain`	Native (CLR)	Debuginfo package or source build
`__libc_start_main`	Native	Debuginfo package

Flame Graphs

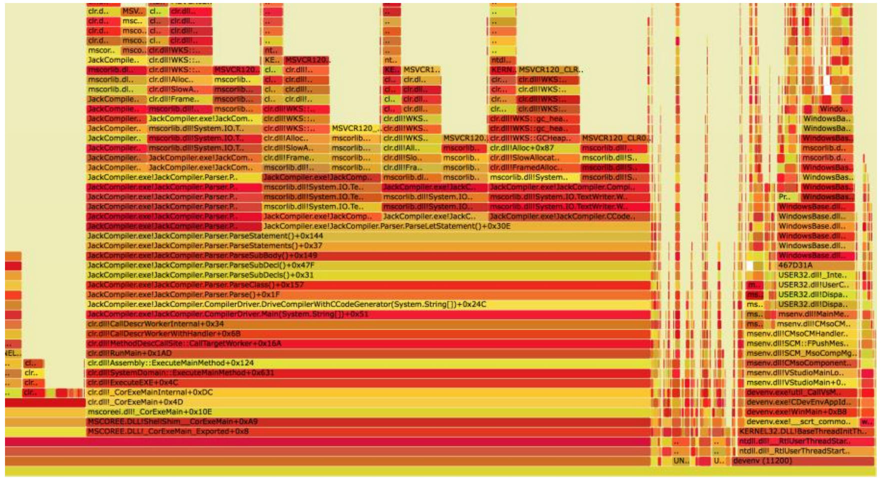
A visualization method (adjacency graph), very useful for stack traces, invented by Brendan Gregg

- http://www.brendangregg.com/flamegraphs.html

Turns thousands of stack trace pages into a single interactive graph

Example scenarios:

- Identify CPU hotspots on the system/application
- Show stacks that perform heavy disk accesses
- Find threads that block for a long time and the stack where they do it



Reading a Flame Graph

.NET DEVELOPER DAYS

Each rectangle is a function

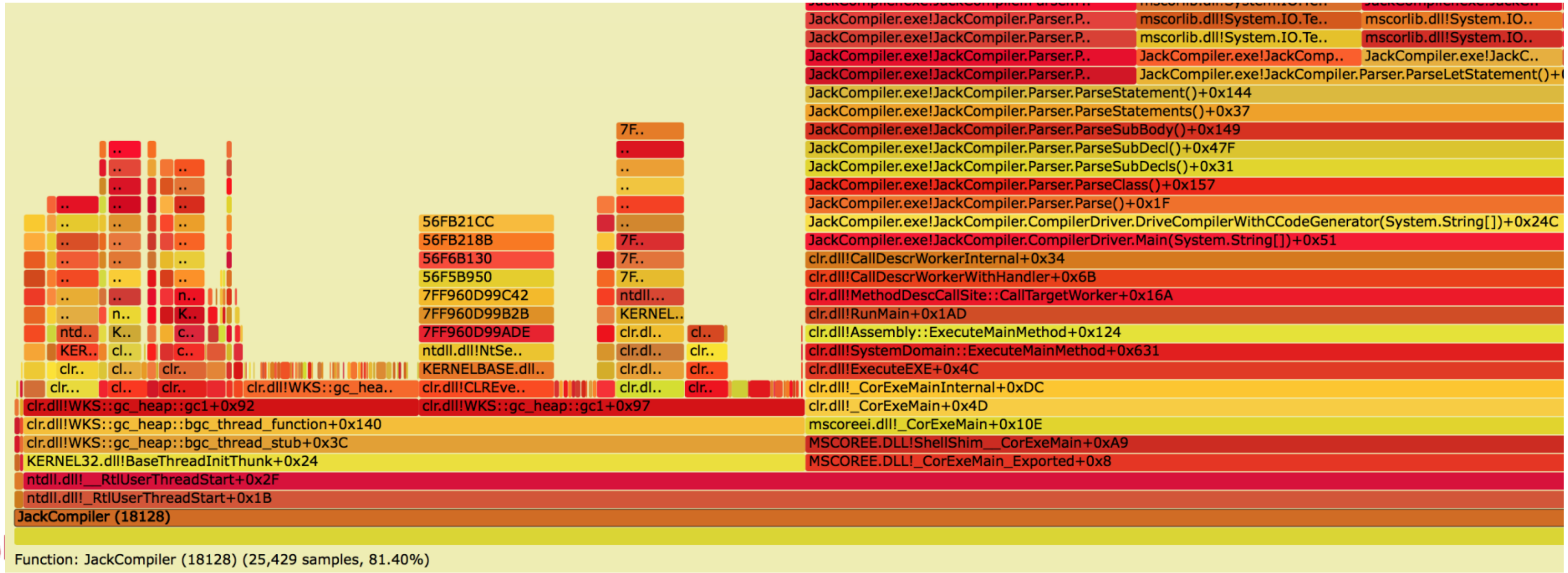
Y-axis: caller-callee

X-axis: sorted stacks (not time)

Wider frames are more common

Supports zoom, find

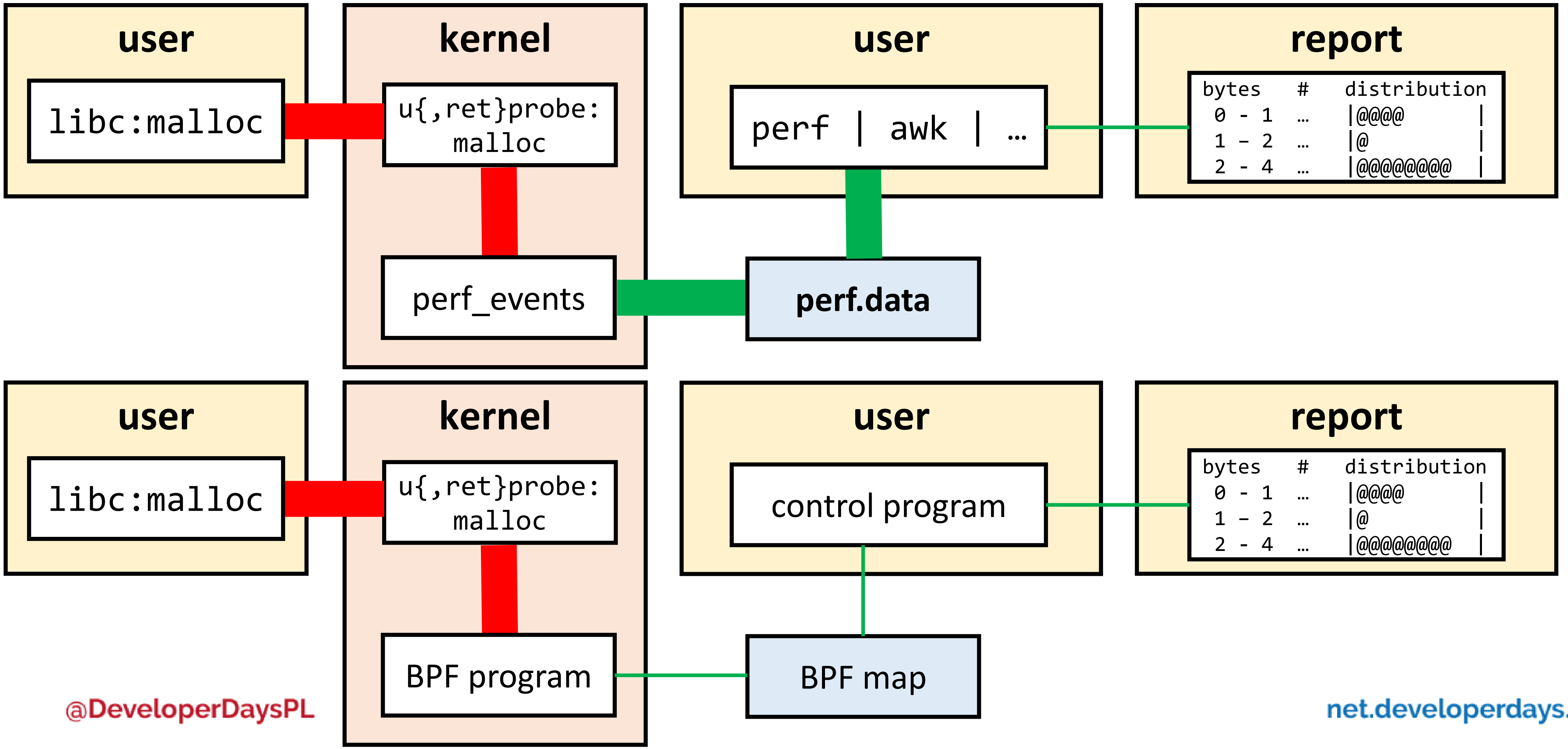
Filter with grep 😎



Demo:
CPU Profiling With Flame Graphs

```
1$ make build && make run
2$ make authbench
3$ make authrecord
```

The Old Way And The New Way: BPF



The BCC BPF Front-End

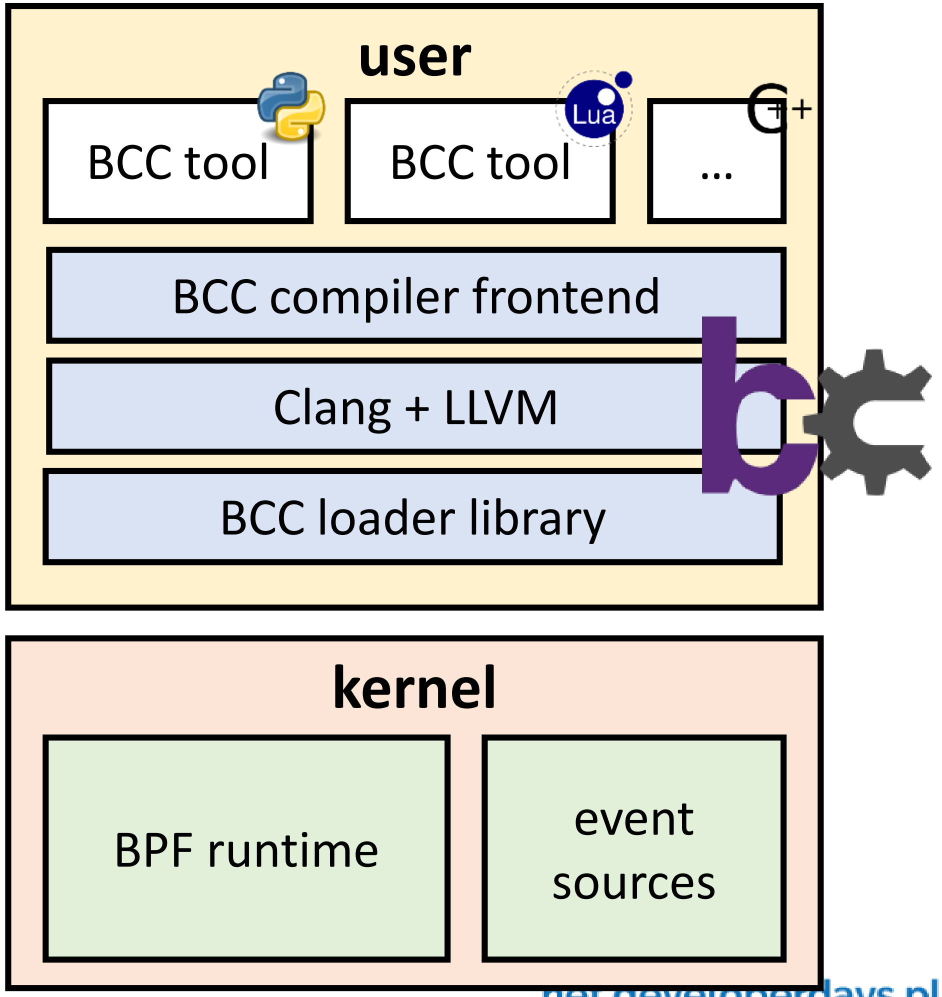
https://github.com/iovisor/bcc

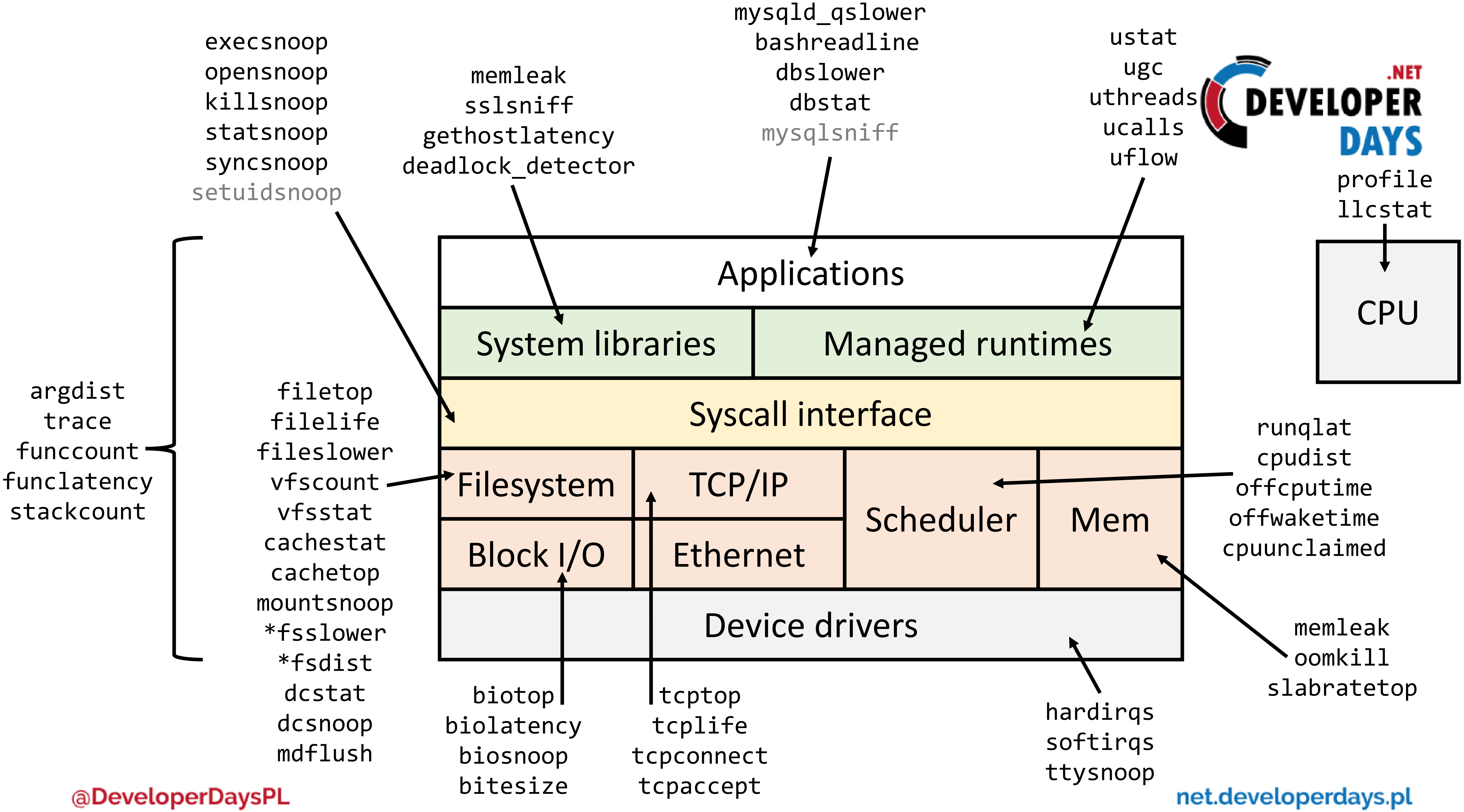
BPF Compiler Collection (BCC) is a BPF frontend library and a massive collection of performance tools

- Contributors from Facebook, PLUMgrid, Netflix, Sela

Helps build BPF-based tools in high-level languages

- Python, Lua, C++

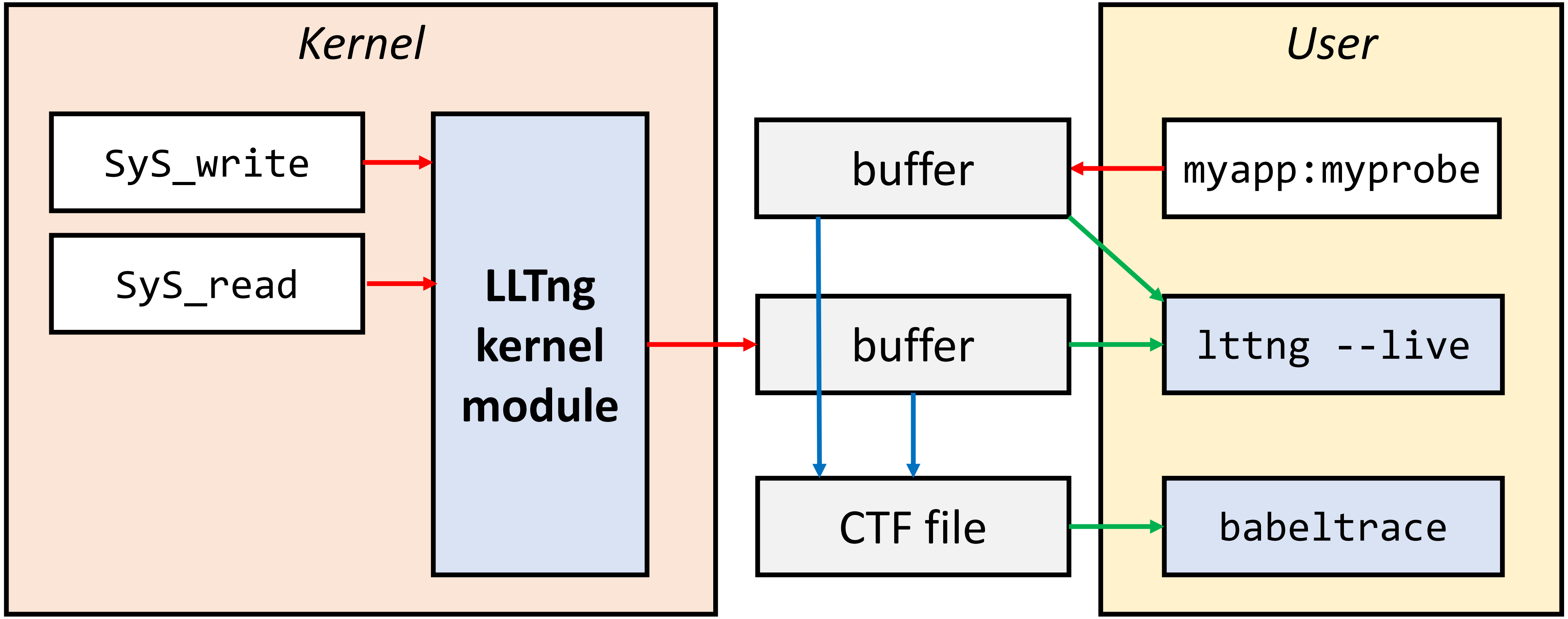



execsnoop
opensnoop
killsnoop
statsnoop
syncsnoop
setuidsnoop
memleak
sslsniff
gethostlatency
deadlock_detector
mysqld_qslower
bashreadline
dbslower
dbstat
mysqlsniff
ustat
ugc
uthreads
ucalls
uflow
.NET DEVELOPER DAYS
profile
llcstat
CPU
Applications
System libraries
Managed runtimes
Syscall interface
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

Demo:
Tracing .NET Core Apps

```
1$ make build && make run
2$ make getstatsbench
3$ make getstatsrecord   3$ make alloccount   3$ make allocstacks
2$ make catsbench        3$ make catsrecord2
```

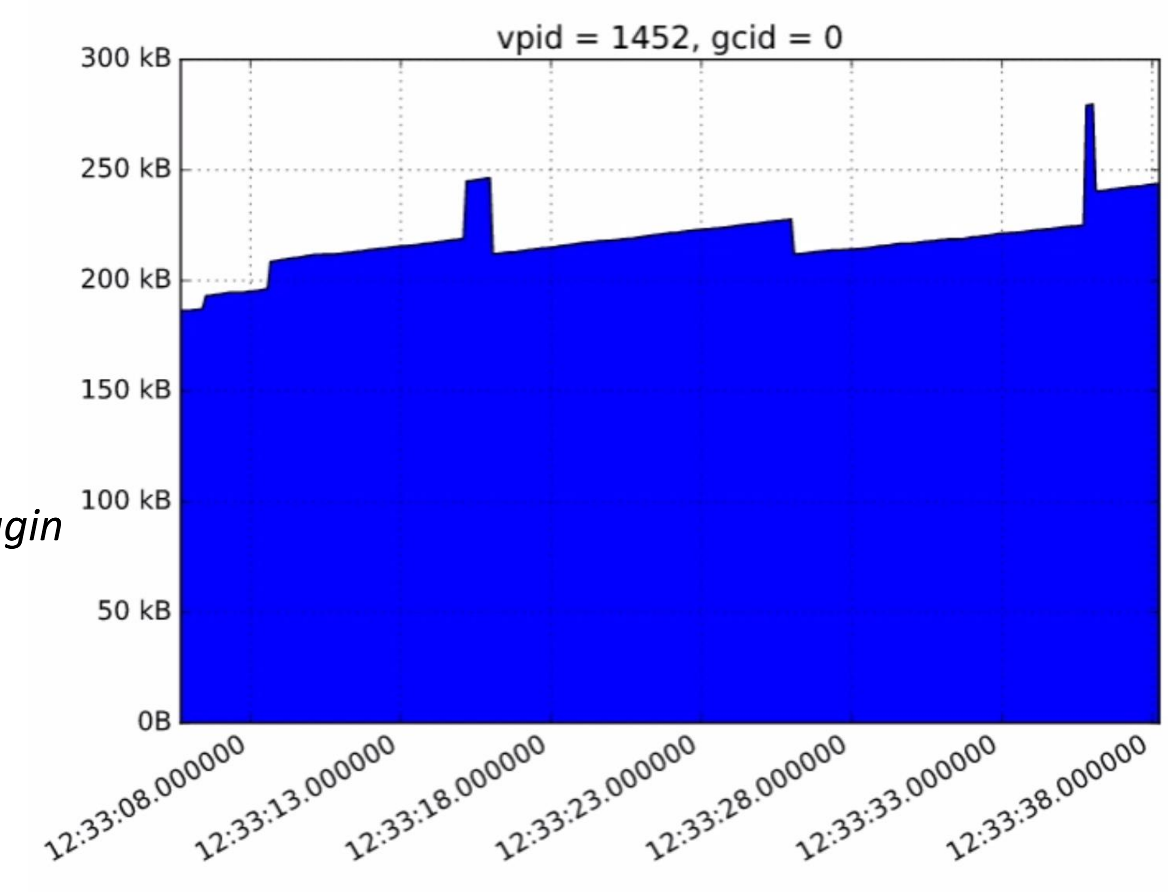
LTTng Architecture



Demo:
Capturing Runtime Events

```
1$ make build && make run
2$ make getstatsbench
3$ make gcrecord  3$ make gcview | grep…  3$ make gcallocstats
```

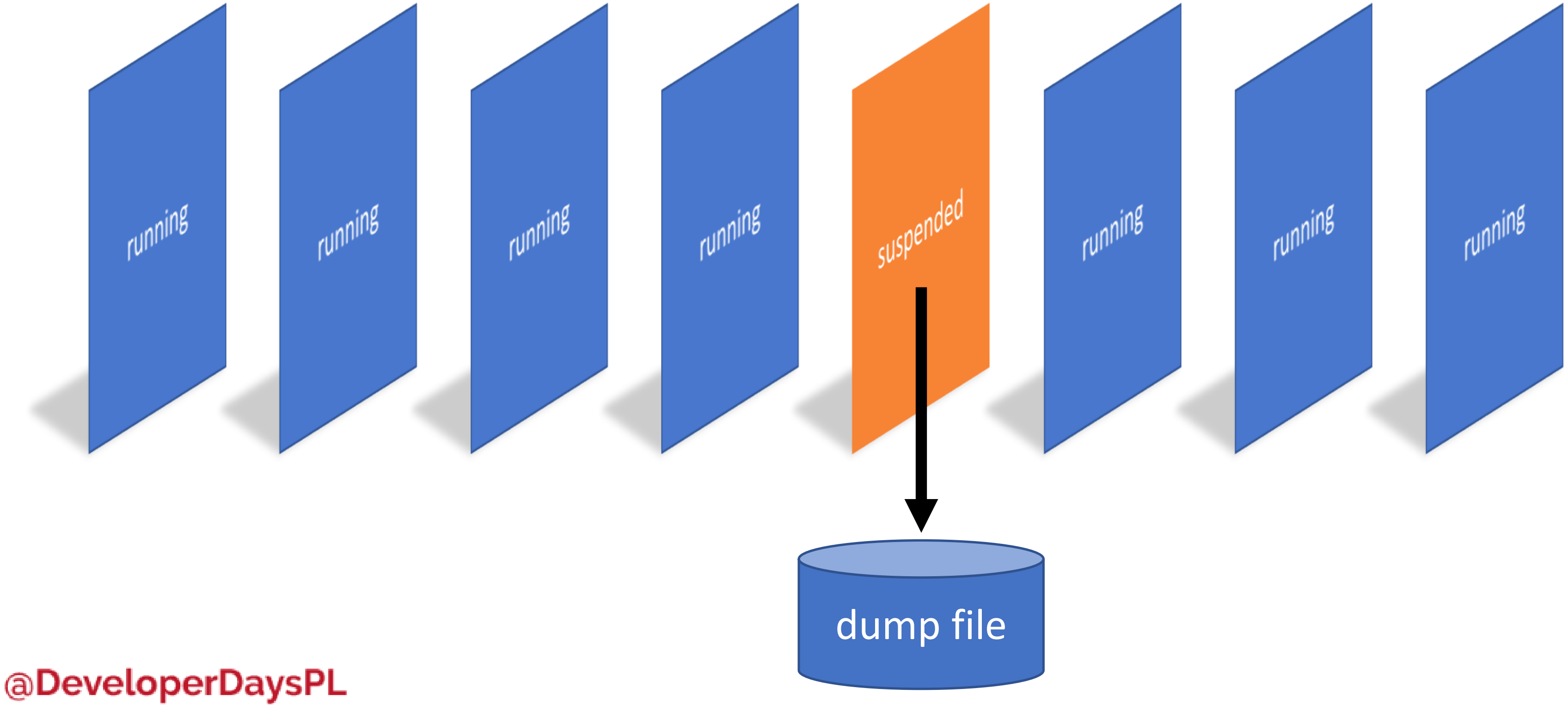
Plotting data from GCStats events, [original work]() by Aleksei Vereshchagin at DotNext Saint Petersburg



Core Dumps

A core dump is a memory snapshot of a running process

Can be generated **on crash** or **on demand**



Generating Core Dumps

/proc/sys/kernel/core_pattern configures the core file name or application to process the crash

ulimit -c controls maximum core file size (often 0 by default)

gcore (part of gdb) can create a core dump on demand

Analyzing .NET Core Dumps

```
$ lldb /usr/bin/dotnet -c core.1788
(lldb) bt
```

```
thread #1: tid = 0, 0x00007f7c3a37c7ef libc.so.6`gsignal + 159, name = 'Buggy', stop reason = signal SIGABRT
* frame #0: 0x00007f7c3a37c7ef libc.so.6`gsignal + 159
  frame #1: 0x00007f7c3a37e3ea libc.so.6`abort + 362
  frame #2: 0x00007f7c399a90bc libcoreclr.so`PROCAbort + 124
  frame #3: 0x00007f7c399a7fbb libcoreclr.so`PROCEndProcess(void*, unsigned int, int) + 235
  frame #4: 0x00007f7c39711318 libcoreclr.so`UnwindManagedExceptionPass1(PAL_SEHException&, _CONTEXT*) + 840
  frame #5: 0x00007f7c397113c9 libcoreclr.so`DispatchManagedException(PAL_SEHException&, bool) + 73
  frame #6: 0x00007f7c39681afa libcoreclr.so`IL_Throw(Object*) + 794
  frame #7: 0x00007f7bc05609e2
  frame #8: 0x00007f7bbfff7d0e
  frame #9: 0x00007f7bbfff9349
  frame #10: 0x00007f7c3971da46 libcoreclr.so`FastCallFinalizeWorker + 6
  frame #11: 0x00007f7c395c0c28 libcoreclr.so`MethodTable::CallFinalizer(Object*) + 600
  frame #12: 0x00007f7c396627de libcoreclr.so`FinalizerThread::DoOneFinalization(Object*, Thread*, int, bool*) + 334
  frame #13: 0x00007f7c396625ea libcoreclr.so`FinalizerThread::FinalizeAllObjects(Object*, int) + 266
  frame #14: 0x00007f7c39662c4e libcoreclr.so`FinalizerThread::FinalizerThreadWorker(void*) + 446
  frame #15: 0x00007f7c395fea62 libcoreclr.so`ManagedThreadBase_DispatchOuter(ManagedThreadCallState*) + 402
  frame #16: 0x00007f7c395ff2be libcoreclr.so`ManagedThreadBase::FinalizerBase(void (*)(void*)) + 94
  frame #17: 0x00007f7c39662ecc libcoreclr.so`FinalizerThread::FinalizerThreadStart(void*) + 204
  frame #18: 0x00007f7c399aad22 libcoreclr.so`CorUnix::CPalThread::ThreadEntry(void*) + 306
  frame #19: 0x00007f7c3afbd6ca libpthread.so.0`start_thread + 202
  frame #20: 0x00007f7c3a44f0af libc.so.6`clone + 95
```

The stuff we care about

libsosplugin.so

```
(lldb) plugin load …/libsosplugin.so
(lldb) setclrpath …
```

DumpObj	Threads
DumpArray	ClrStack
DumpHeap	EEHeap
GCRoot	DumpDomain
PrintException	DumpAssembly

Demo:
Dump Generation And Analysis

```
1$ make dockersvc && make dockerrun
2$ make update
3$ make updatelogs    3$ make updateanalyze
```

Checklist: Preparing Your Environment

- ❑`export COMPlus_PerfMapEnabled=1`
- ❑AOT perf map with crossgen
- ❑Debuginfo package for libcoreclr, libc
- ❑Install perf/BCC tools
- ❑`export COMPlus_EnableEventLog=1`
- ❑`ulimit -c unlimited` (or managed by system)
- ❑Install gdb (for gcore), lldb-3.x

Summary

We have learned:

✓To profile CPU activity in .NET Core apps

✓To visualize stack traces (e.g. of CPU samples) using flame graphs

✓To use Linux tracing tools with .NET Core processes

✓To capture .NET Core runtime events using LTTng

✓To generate and analyze core dumps of .NET Core apps

References

perf and flame graphs

- https://perf.wiki.kernel.org/index.php/Main_Page
- http://www.brendangregg.com/perf.html
- https://github.com/brendangregg/perf-tools

.NET Core diagnostics docs

- https://github.com/dotnet/coreclr/blob/master/Documentation/project-docs/linux-performance-tracing.md
- https://github.com/dotnet/coreclr/blob/master/Documentation/building/debugging-instructions.md

My blog posts

- http://blogs.microsoft.co.il/sasha/2017/02/26/analyzing-a-net-core-core-dump-on-linux/
- http://blogs.microsoft.co.il/sasha/2017/02/27/profiling-a-net-core-application-on-linux/
- http://blogs.microsoft.co.il/sasha/2017/03/30/tracing-runtime-events-in-net-core-on-linux/

BCC tutorials

- 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

Thank You!

Slides: https://s.sashag.net/ddwaw17-3

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

Sasha Goldshtein

CTO, Sela Group

goldshtn

goldshtn

Please rate this session using

Developer Days Mobile App

or at

login.developerdays.pl

Strategic Sponsors

KMD
KMDPOLAND.PL

7N

Gold Sponsors

sonova
HEAR THE WORLD

VOLVO
Volvo Group IT

Relativity

ABB

Silver Sponsors

JCommerce

RAVENDB

it KONTRAKT

DevExpress

Millennium bank

IT outsourcing nbc

labs
software specialists

ASPOSE
File Format APIs

Demant
Technology Centre

LUXOFT