

501案例篇：動態追蹤怎麼用？(上)

Hazel Shen



效能調整方法

- GDB 設置應用程式斷點
- 在程式碼裡面添加 tag, debugger
- 從日誌 log 裡面挖資訊,



動態追蹤

- 通過探針（Probe）機制查詢，探針又分為三種
 - 靜態探針
 - 動態探針
 - 硬體事件
- 性能消耗小：5% 以內
- 工具例子：ftrace / perf / systemtap / eBPF / BCC / sysdig / strace



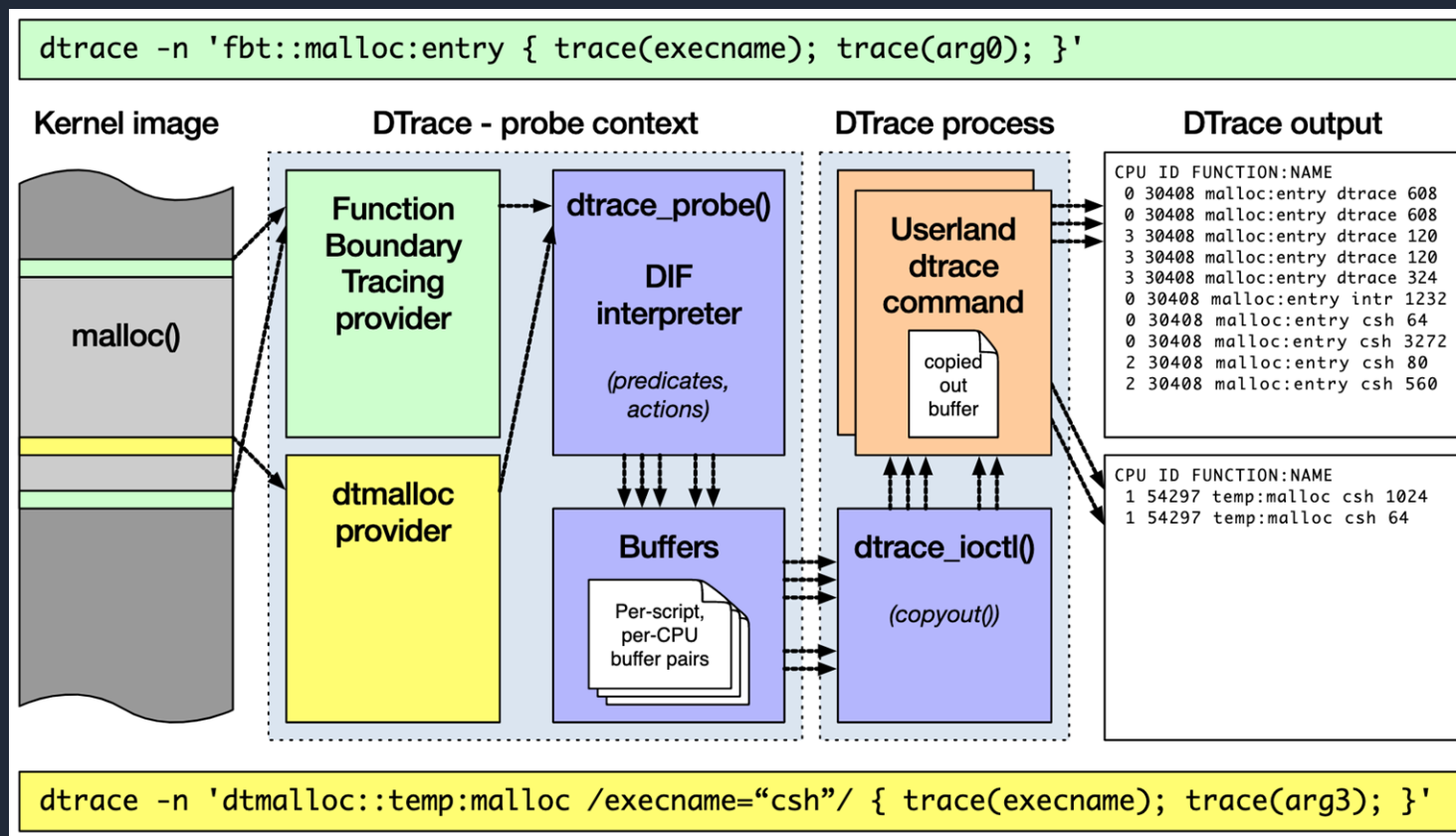
動態追蹤編年史

年份	技術
2004	kprobes/kretprobes
2008	ftrace
2005	systemtap
2009	perf_events
2009	tracepoints
2012	uprobes
2015 ~ 至今	eBPF (Linux 4.1+)



Source: https://blog.arstercz.com/introduction_to_linux_dynamic_tracing/

動態追蹤的鼻祖 – DTrace of Solaris



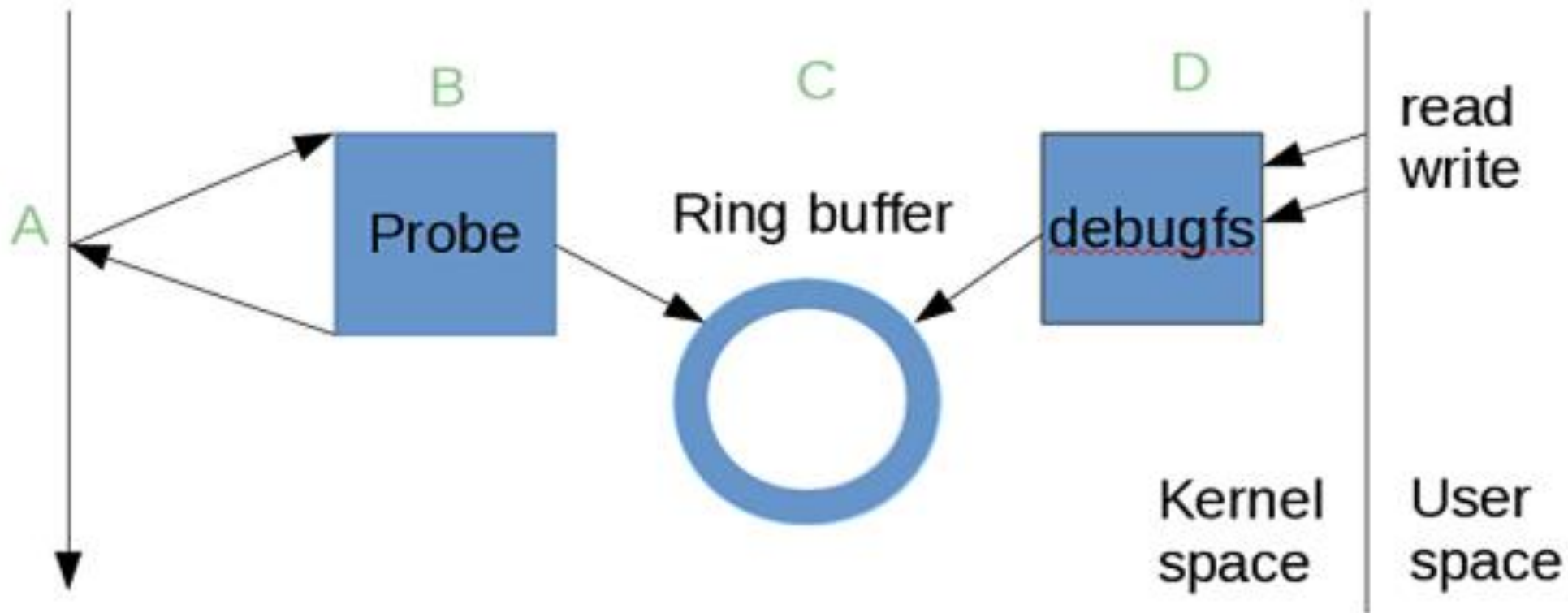
ftrace

- 最早用於函數追蹤
- 類似 procfs
- 提供多個追蹤器，可追蹤不同類型的訊息 e.g 調用函數、中斷關閉、Process 調度
- `cd /sys/kernel/debug/tracing`



```
[hazel@bastion ~]$ sudo ls /sys/kernel/debug/tracing
README
available_events
available_filter_functions
available_tracers
buffer_size_kb
buffer_total_size_kb
current_tracer
dyn_ftrace_total_info
enabled_functions
events
free_buffer
function_profile_enabled
hwlat_detector
instances
kprobe_events
kprobe_profile
max_graph_depth
options
per_cpu
printk_formats
saved_cmdlines
saved_cmdlines_size
saved_tgids
set_event
set_event_pid
set_ftrace_filter
set_ftrace_notrace
set_ftrace_pid
set_graph_function
set_graph_notrace
snapshot
stack_max_size
stack_trace
stack_trace_filter
synthetic_events
timestamp_mode
trace
trace_clock
trace_marker
trace_marker_raw
trace_options
trace_pipe
trace_stat
tracing_cpumask
tracing_max_latency
tracing_on
tracing_thresh
uprobe_events
uprobe_profile
```

Ftrace 原理



ftrace

- echo do_sys_open > set_graph_function
- echo function_graph > current_tracer
- echo funcgraph-proc > trace_options
- echo 1 > tracing_on
- ls
- echo 0 > tracing_on
- cat trace

設置要跟蹤的函數

設定跟蹤選項，開啟函數呼叫跟蹤

跟蹤 process

開啟跟蹤

執行 ls 命令

關閉跟蹤

查看跟蹤結果

ftrace

```
0) 0.484 us | /
6) | do_filp_open() {
6) | path_openat() {
6) | alloc_empty_file() {
6) | __alloc_file() {
6) | kmem_cache_alloc() {
6) | _cond_resched() {
6) 0.021 us | rcu_all_qs();
6) 0.169 us | }
6) 0.025 us | should_failslab();
6) 0.045 us | memcg_kmem_get_cache();
6) 0.027 us | memcg_kmem_put_cache();
6) 0.822 us | }
6) 0.025 us | security_file_alloc();
6) 0.024 us | __mutex_init();
6) 1.287 us | }
6) 1.439 us | }
```

trace-cmd - 打包過後的 ftrace

- `sudo dnf install trace-cmd`
- `trace-cmd record -p function_graph -g do_sys_open -O funcgraph-proc ls`

```
y: 0.000 us | do_sys_open(0, "/dev/null", O_RDONLY, 0);
: 0.186 us | }
: <...>-1206082 [005] 17176589.936352: funcgraph_exit
y: 0.035 us | putname() {
: 0.035 us | kmem_cache_free();
: 0.190 us | }
: <...>-1206082 [005] 17176589.936353: funcgraph_exit
: + 37.356 us | }
```

Perf 的原理

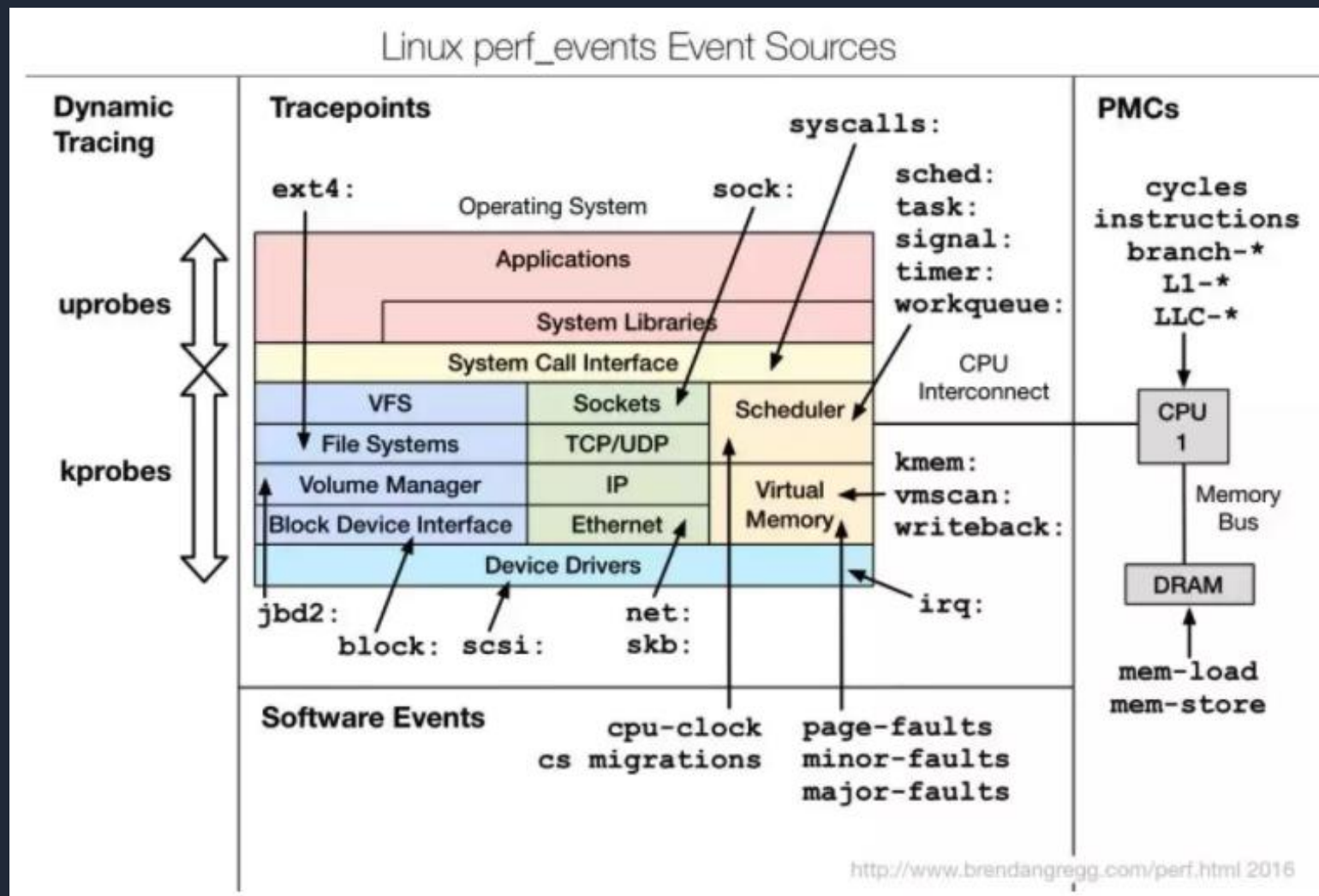
- Linux 2.6.31 後內建的效能分析工具，並隨著核心 released
- 可以利用 PMU (Performance Monitoring Unit), tracepoint, 以及核心內部的特殊計數器(counter)來進行統計
- 針對目標進行取樣，紀錄特定條件下偵測的事件是否發生、發生頻率



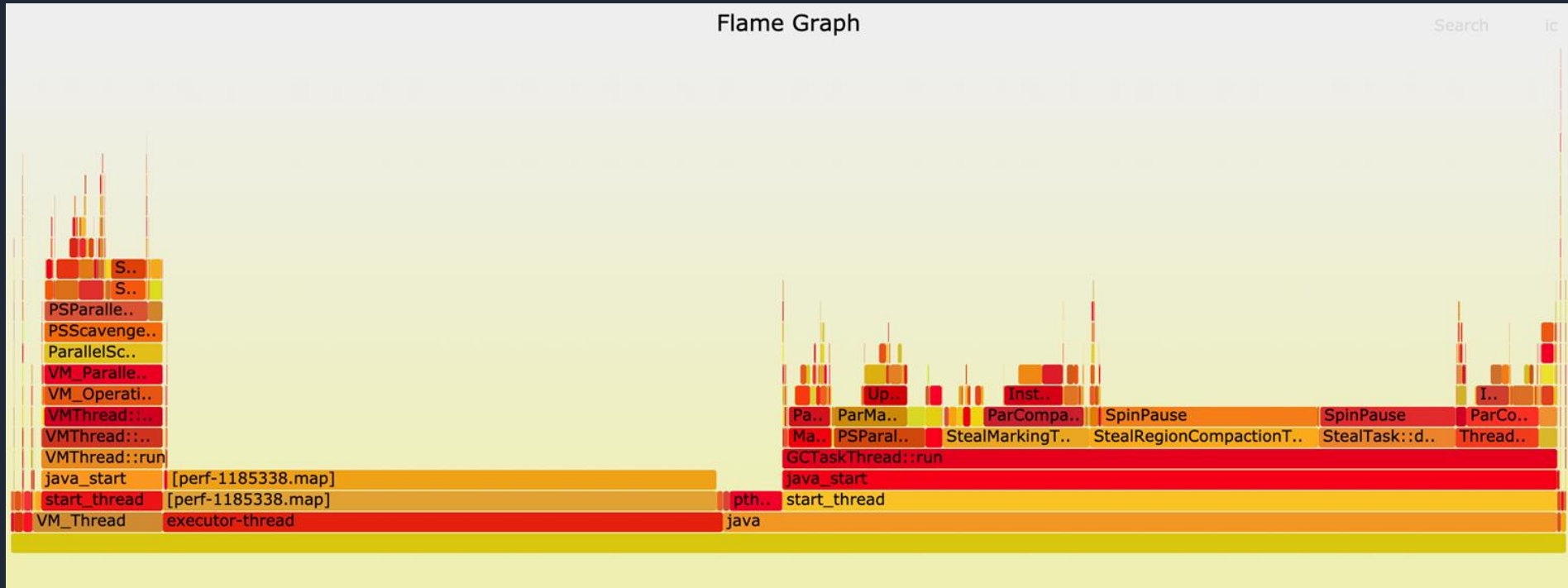
Perf – Performance Event

Perf 能觸發的事件：

- hardware
- software
- tracepoint



FlameG



```
perf script > out.perf
```

```
# Dump perf.data
```

```
git clone --depth 1 https://github.com/brendangregg/FlameGraph.git
```

折疊報告

```
FlameGraph/flamegraph.pl out.folded > out.svg
```

產生火焰圖

小結

- 在 Linux 世界中，常用的動態追蹤方法：ftrace / perf / eBPF / SystemTap
- 當已經定位某個 kernel 函數，可以用 ftrace 追蹤執行過程



思考題

1. Perf / 火焰圖 和 ftrace 有什麼不一樣的地方？
2. 當需要了解 kernel 行為時，如何根據適合的使用場景做選擇



511案例篇：動態追蹤怎麼用？(下)

Hazel Shen



perf

- 針對事件進行採樣、評估函數調用頻率
- 除了以上函數相關分析，perf 還可以分析以下硬體事件：
 - CPU cache
 - CPU task migration
 - 分支預測
 - 指令週期



perf 指令

perf list - 查詢所有支持的 events


perf probe - 動態添加想觀察的事件

perf trace ls - 系統跟蹤



案例一：Perf probe 實際操作 kernel 函數

- `sudo perf probe - -add do_sys_open` #添加 do_sys_open 探針
- `sudo perf record -e probe:do_sys_open -aR sleep 10` #針對十秒內採樣
- `sudo perf script` # 查看採樣結果



```
sleep 1992793 [003] 17411499.255656: probe:do_sys_open: (ffffffffb3ac38d0)
sleep 1992793 [003] 17411499.255663: probe:do_sys_open: (ffffffffb3ac38d0)
sleep 1992793 [003] 17411499.255664: probe:do_sys_open: (ffffffffb3ac38d0)
sleep 1992793 [003] 17411499.255669: probe:do_sys_open: (ffffffffb3ac38d0)
redis-server 1379112 [001] 17411499.287763: probe:do_sys_open: (ffffffffb3ac38d0)
redis-server 1379112 [001] 17411499.387927: probe:do_sys_open: (ffffffffb3ac38d0)
in:imjournal 4187061 [003] 17411499.475419: probe:do_sys_open: (ffffffffb3ac38d0)
redis-server 1379112 [001] 17411499.488300: probe:do_sys_open: (ffffffffb3ac38d0)
redis-server 1379112 [001] 17411499.588704: probe:do_sys_open: (ffffffffb3ac38d0)
redis-server 1379112 [001] 17411499.689008: probe:do_sys_open: (ffffffffb3ac38d0)
```

案例一：Perf probe - process 到底在打開哪些文件？

- `perf probe - -del probe:do_sys_open` # 先刪除舊的探針
- `perf probe --add 'do_sys_open filename:string'` # 添加帶有參數的探針
- `perf record -e probe:do_sys_open -aR ls` # 重新採樣
- `perf script` # 查看結果
- `perf probe - -del probe:do_sys_open` # 刪除探針



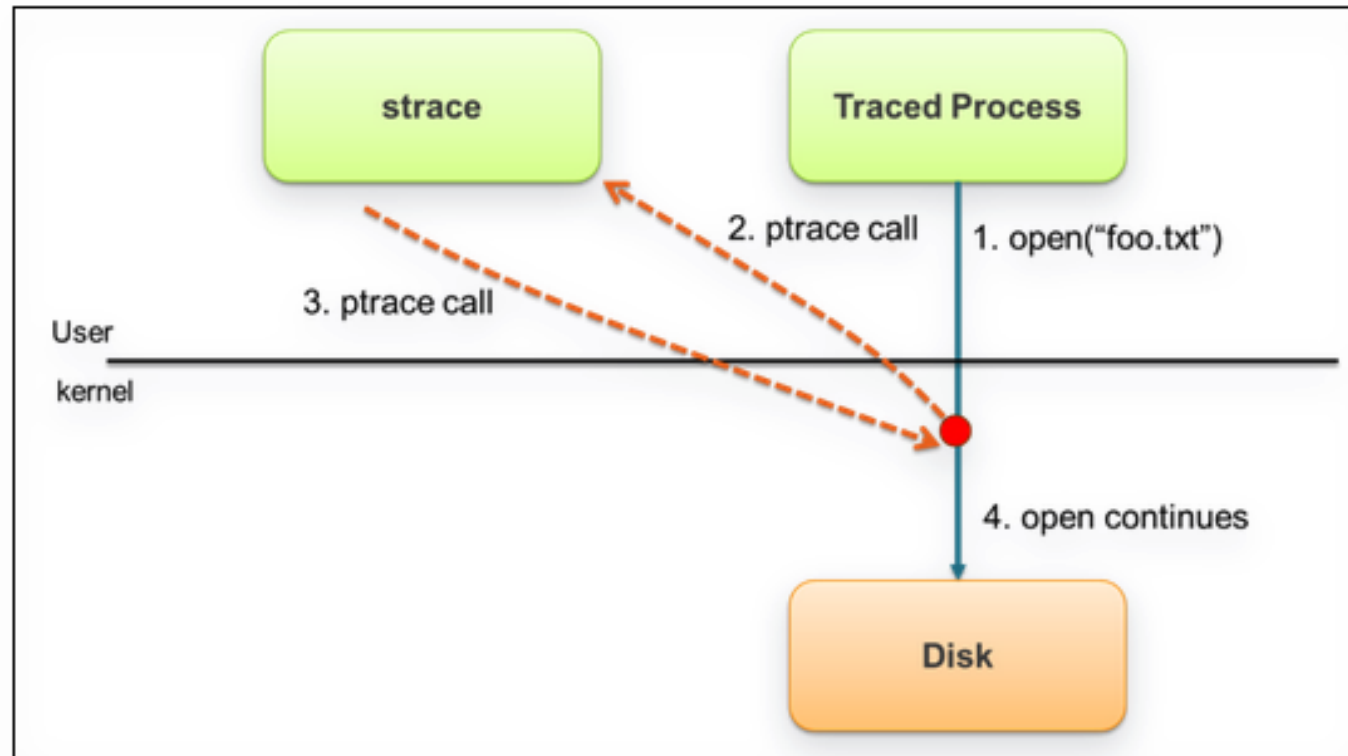
Strace – 基於系統調用 ptrace

- `strace -T -ttt -p pid`
- `strace ls`
- 好用但對 process performance 有影響：user / kernel 狀態切換
- SIGSTOP的中斷影響到效能
- 在需要 performance 的情境（e.g. 資料庫），作者並不推薦



Strace 運作原理

How does strace work?



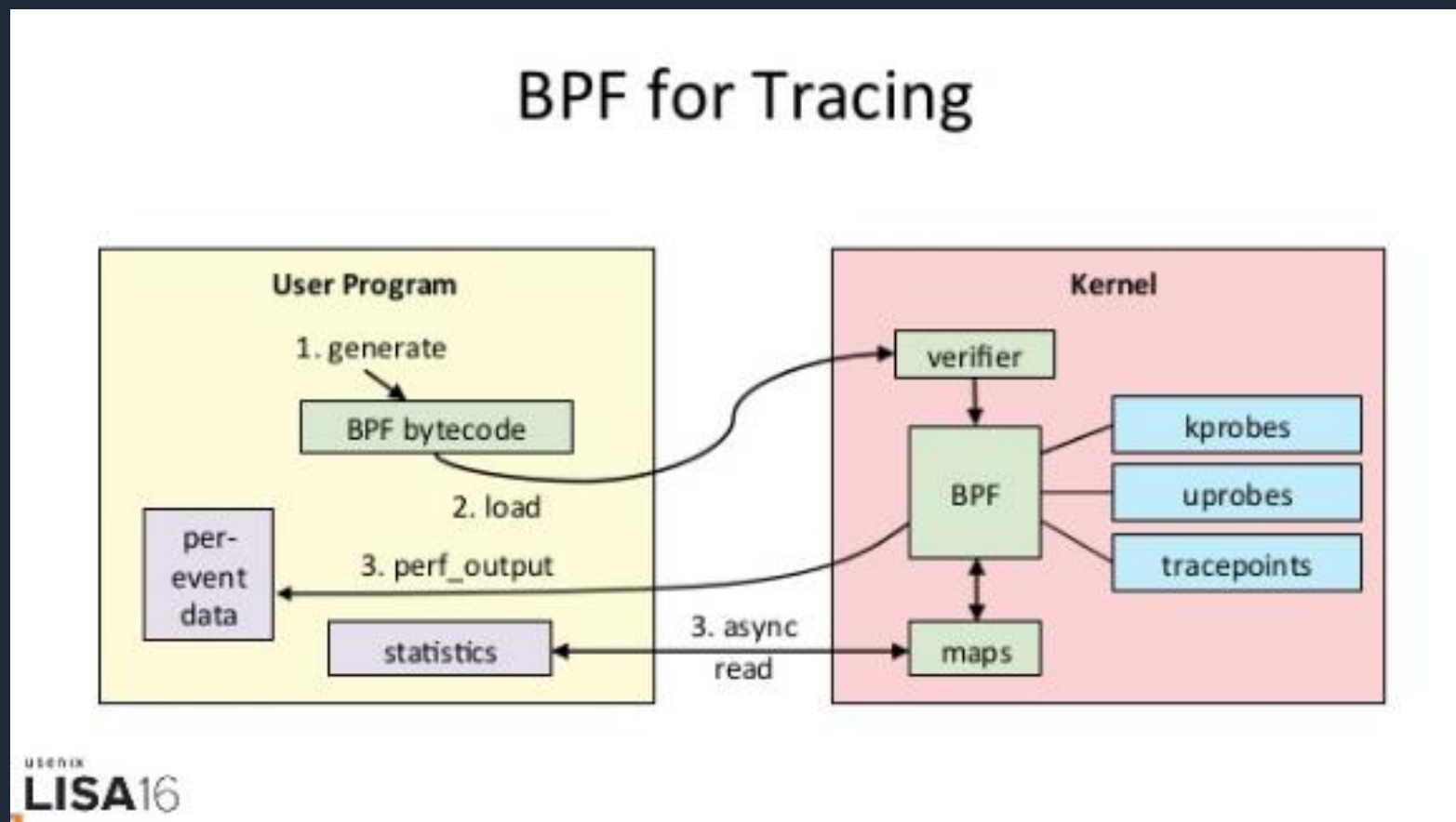
Strace 簡單的讀寫測試

- `dd if=/dev/zero of=/dev/null bs=1 count=500k`
- `strace -eaccept dd if=/dev/zero of=/dev/null bs=1 count=500k`



eBPF & BCC (BPF Compiler Collection)

- ftrace / perf 無法通過 script 自由擴增
- eBPF 即 Linux 版的 Dtrace，可以透過 C 語言自由擴增
- 把過程透過 Python 抽象化，BCC 就是成品



BCC (BPF Compiler Collection)

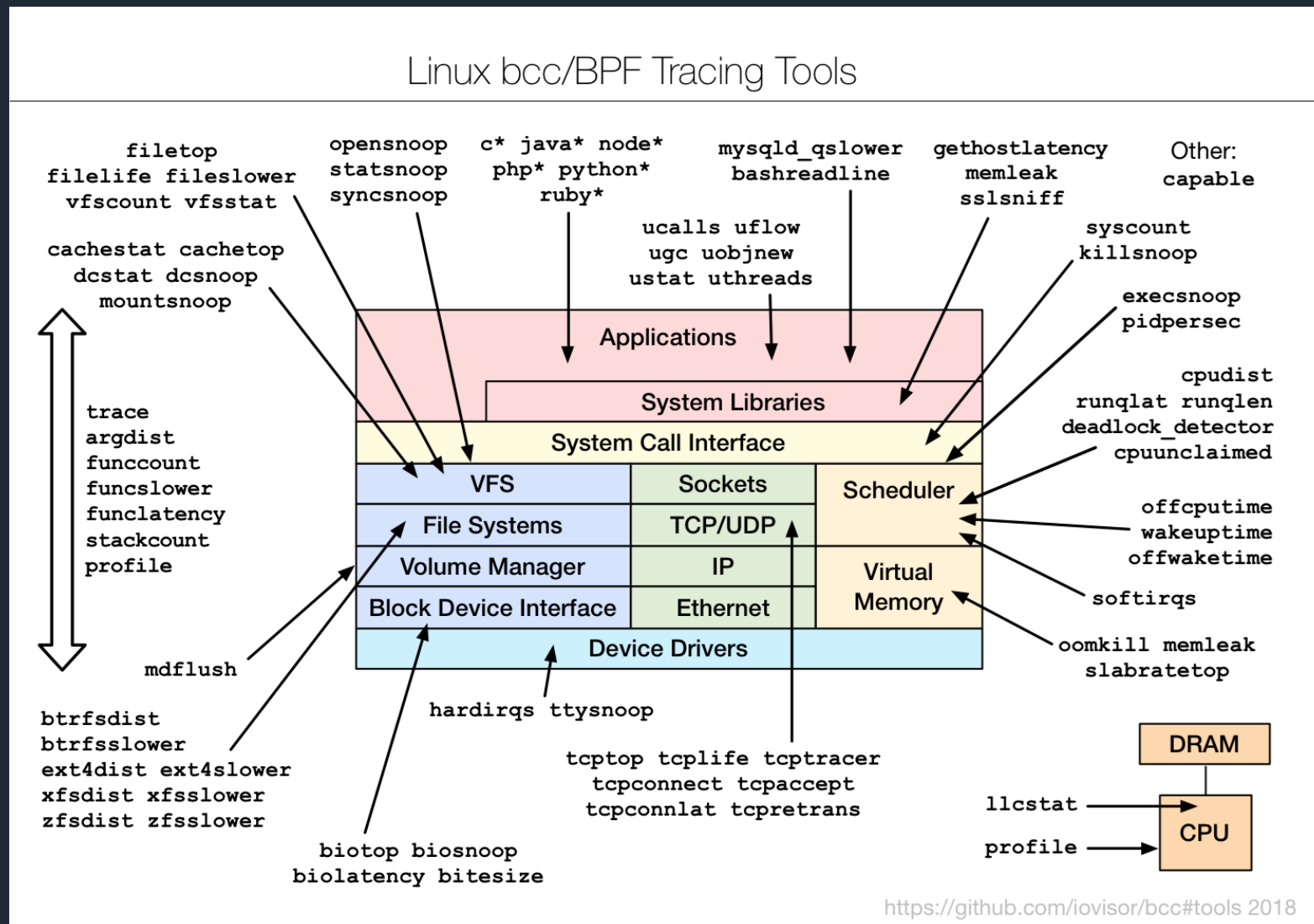
- eBPF 的 event sources (e.g kprobe, uprobe, tracepoint) / data 操作(Maps) 轉換成 Python 的接口、並且支持 Lua
- 核心事件的邏輯處理，還是需要使用 C Language 撰寫
- `yum install bcc-tools`



Source: <https://www.redhat.com/en/blog/introduction-ebpf-red-hat-enterprise-linux-7>
有介紹 eBPF tools 要怎麼開發

BCC 內建的 Tools

- /usr/share/bcc/tools



BCC Tools - Developing

- trace-open.py
- 使用門檻相較 ftrace / perf 來得高

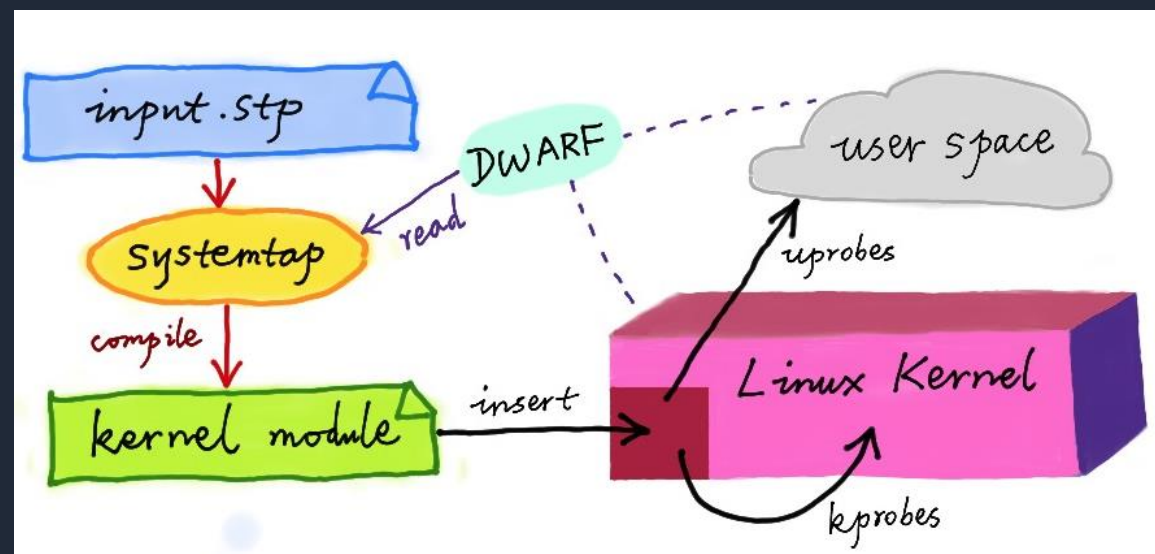
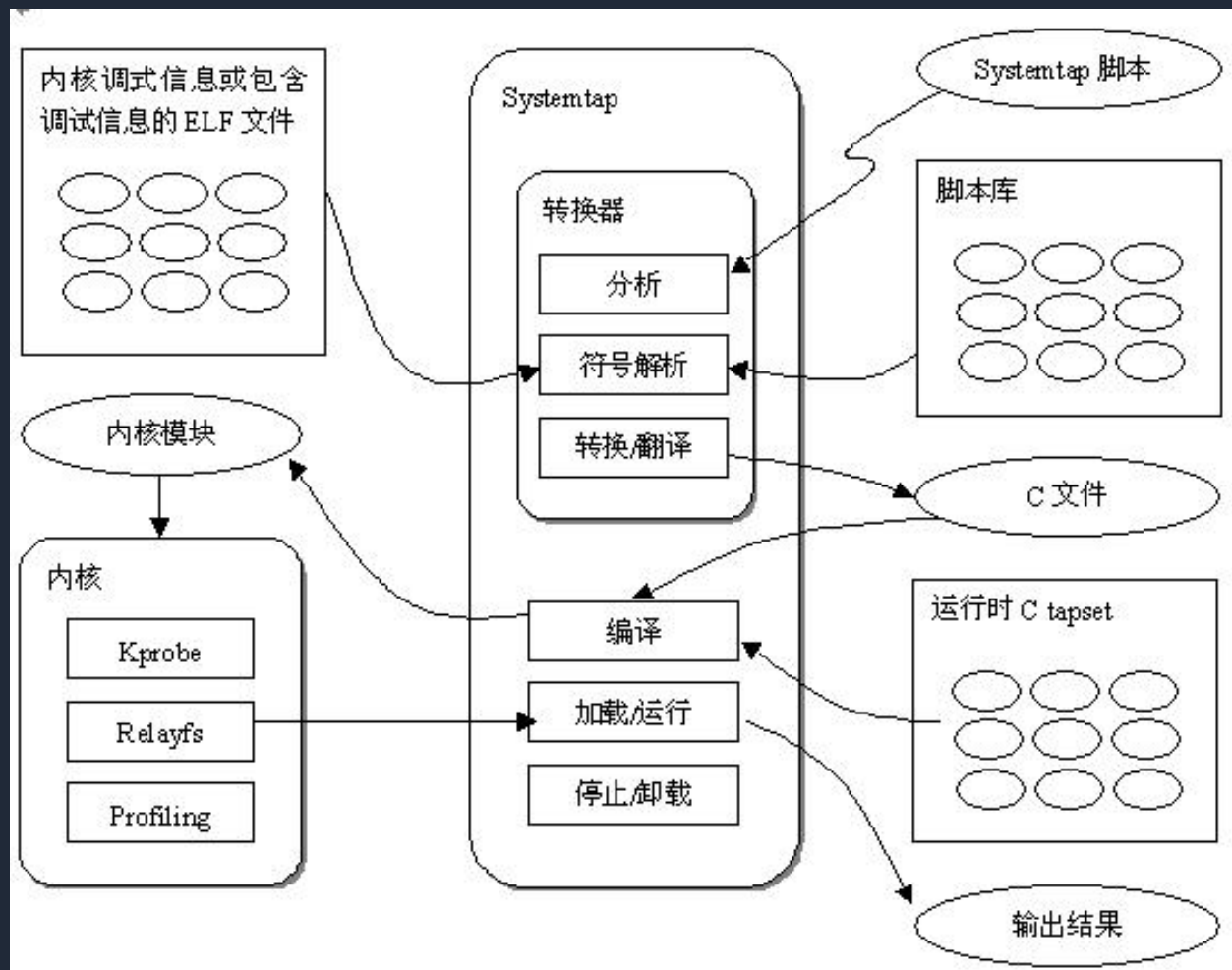


SystemTap

- Linux 中，功能最接近 DTrace 的動態追蹤機制
- 游離於Kernel 之外，相較於 eBPF 根植與Kernel 中
- 只在 RHEL 系統中好用（抱券
- 優點：支援 Linux 3.X 等舊版本



SystemTap



Sysdig – 隨著容器普及而誕生

- 主要用於容器的動態追蹤
- `sysdig = strace + tcpdump + htop + iftop + lsof + docker inspect`
- Kernel 版本 ≥ 4.14 ，可透過 eBPF 增加擴展

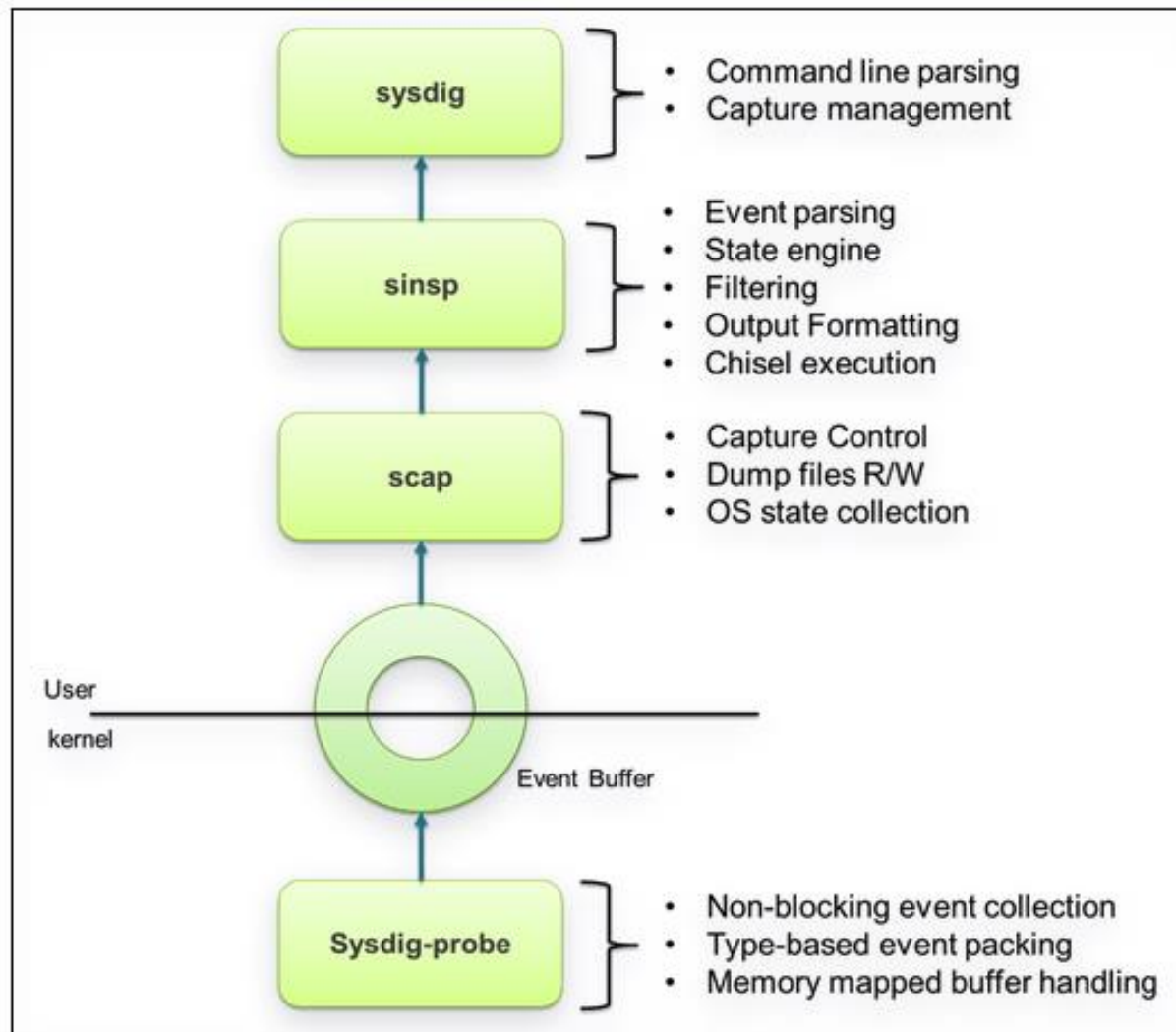


Sysdig 原理

Kernel module 模式獲取所有的 system call

使用方式類似 libpcap / tcpdump

How does sysdig work? Ok, now the fun part - let's talk about sysdig.



常見動態追蹤場景 / 工具

使用場景	推薦工具
Kernel 函數追蹤 (e.g kprobes)	ftrace, trace-cmd
CPU 性能分析和 stack 解析	perf, 火焰圖
RHEL 動態跟蹤內核或者應用程序函數事件	SystemTap
4.X 內核跟蹤內核或應用程序函數或事件	ebpf, bcc-tools
Docker 容器應用性能分析	sysdig

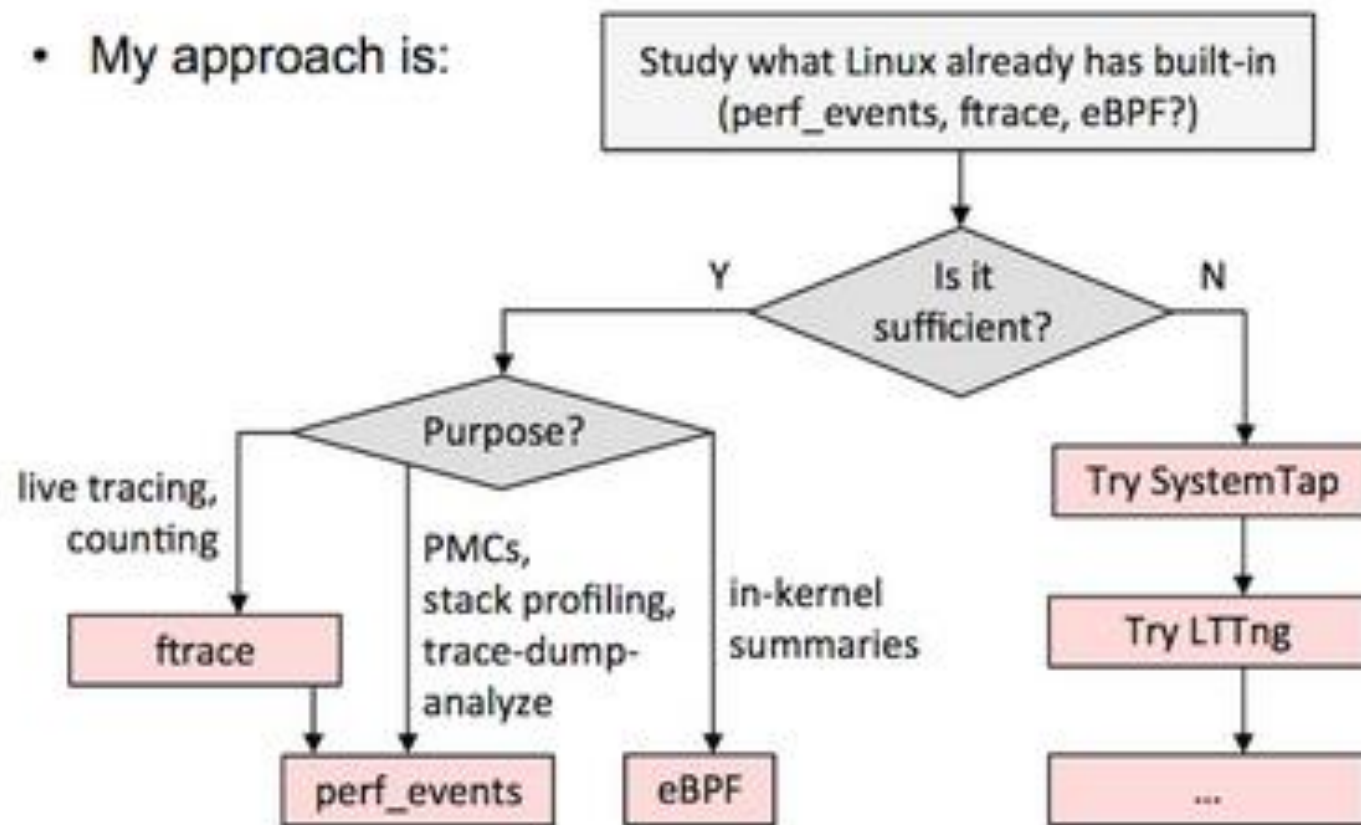
小結

- 大多數的情況下：perf 配合火焰圖
- 新版 Kernel：eBPF & BCC
- 舊版 Kernel：systemTap (eBPF 支援有限)
- 使用動態追蹤技術時，會需要 Kernel Symbol Table (內核符號表) 以得到更詳盡的分析訊息、加速動態追蹤的過程



Brendan Gregg 的工具使用判斷

- My approach is:



問題思考

- 實際環境中用的動態追蹤？



謝謝您的寶貴時間

- [Introduction of eBPF](#)
- [從 Cilium 認識 cgroup ebpf](#)
- [年末大聚會！歡迎來分享！](#)

