Do It Live

Measuring Your Applications in Production

Jason Keene

Pivotal Software

CLOUDFOUNDRY

kubernetes

Measuring your workloads as they are running in a production environment is invaluable for a developer

Why bother measuring in production?

- Observe your software under load
- See faults as they occur
- Discover patterns of usage of your users
- Debug problems:
  - Reproducing the problem in an artificial environment is too difficult or time consuming
  - You simply do not know how to reproduce the problem

Ultimately

Production is Reality

Everything else is at best a proximity

You need to be able to debug problems in production,

but more importantly

Understanding the character of your workloads
is critical to their successful operation.

The more you understand the software you are
running the more successful you will be at running it.

Debugging is not merely the act of making bugs go away. It is the act of **understanding** and **gaining new knowledge** about the way the system works.

- Bryan Cantrill (goto; 2017)

Solve Problems

&&

Understand our Software

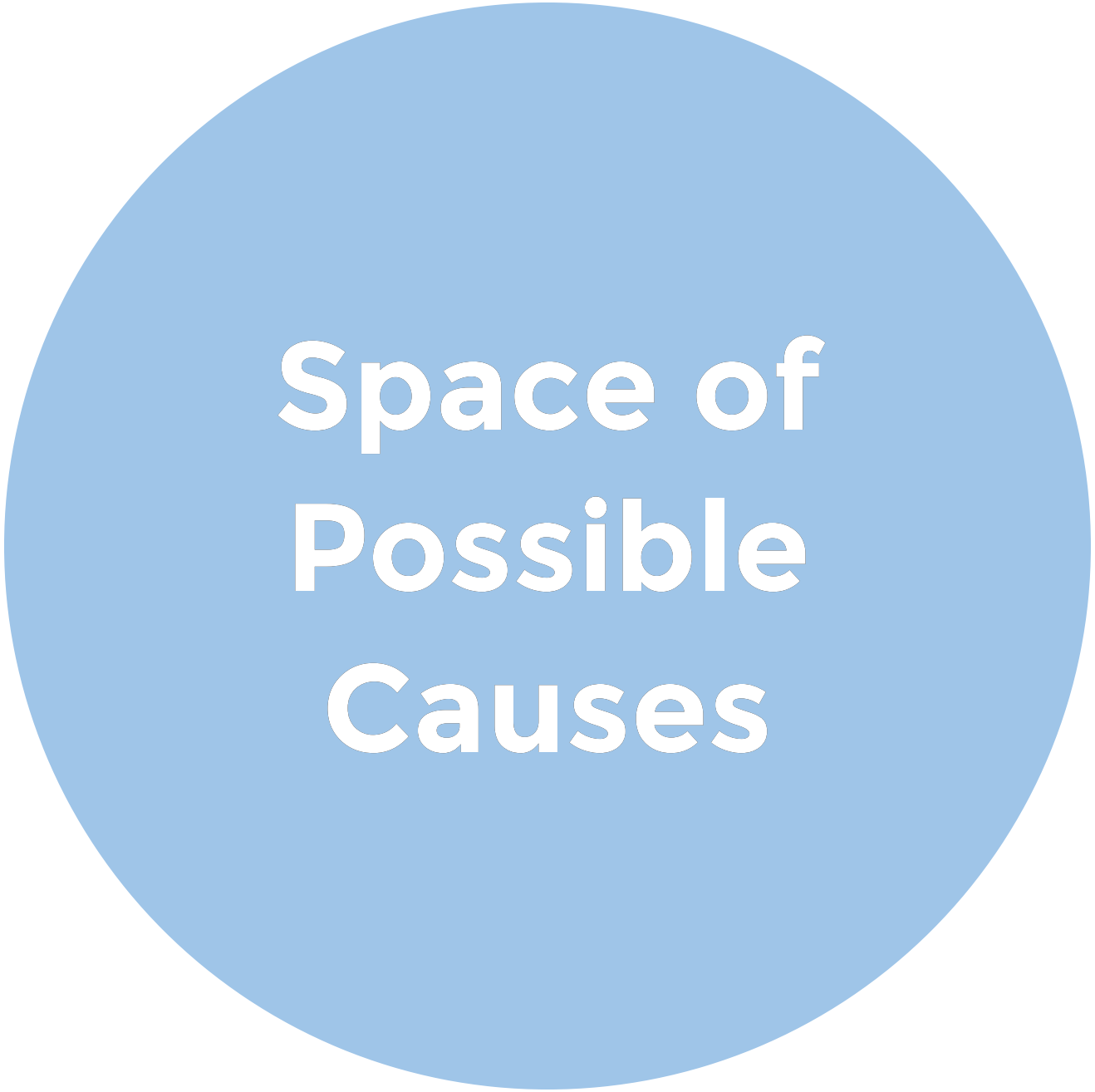
Method
Tools
Practice

Method

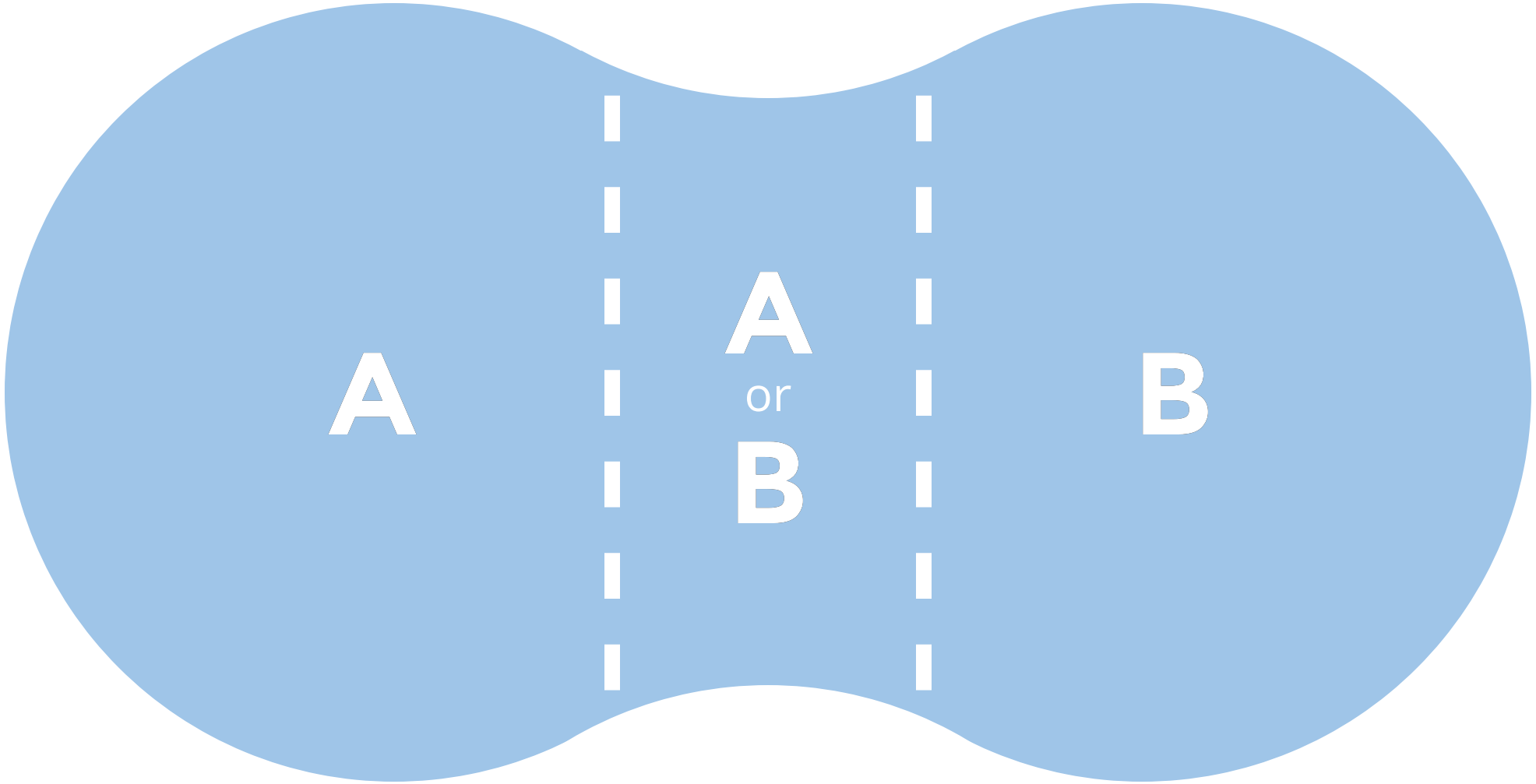
Tools

Practice

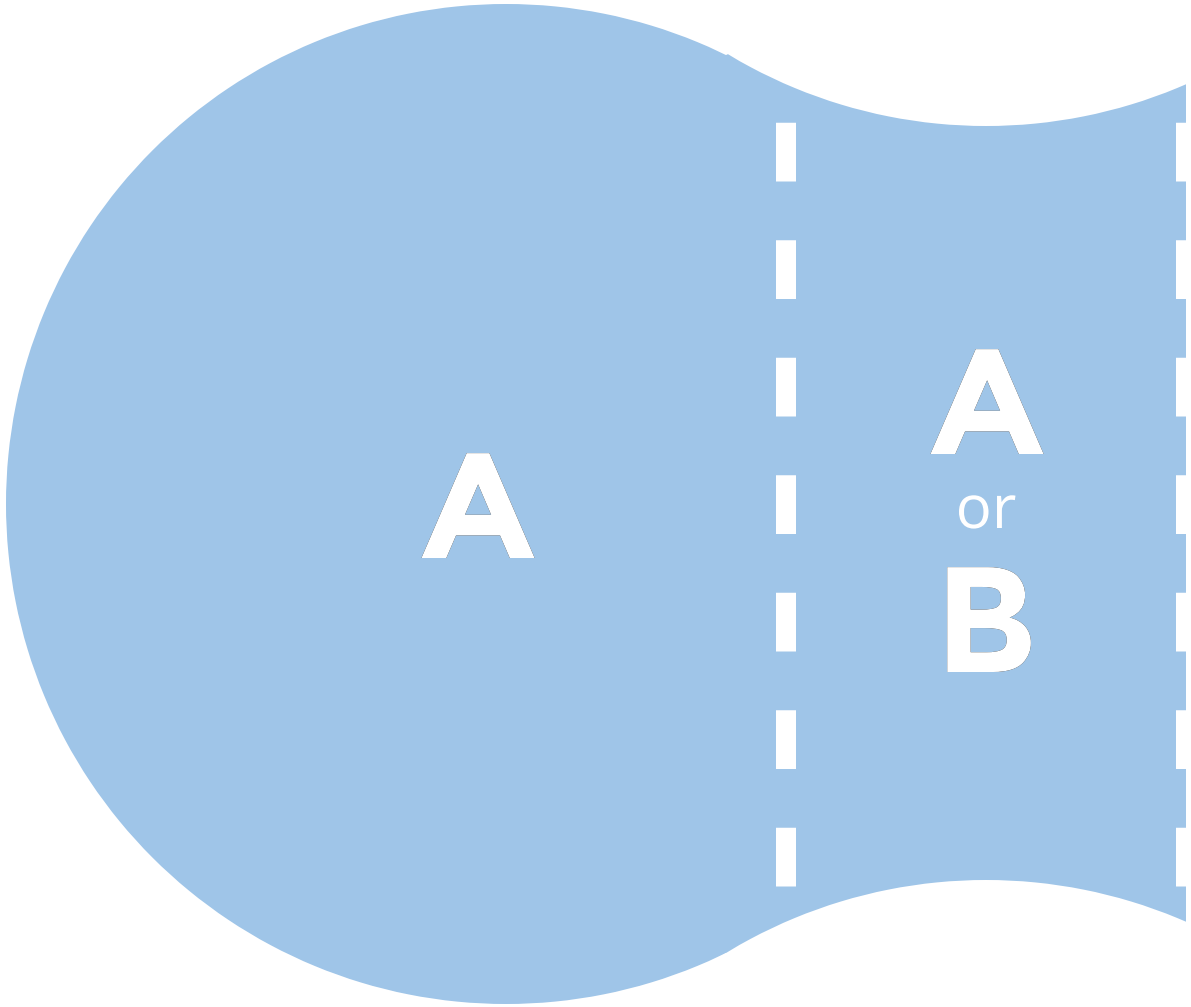
Ask **Questions**, Get **Answers**


Space of
Possible
Causes

A **Question** Divide the Possibility Space



An **Answer** Eliminates Possibilities

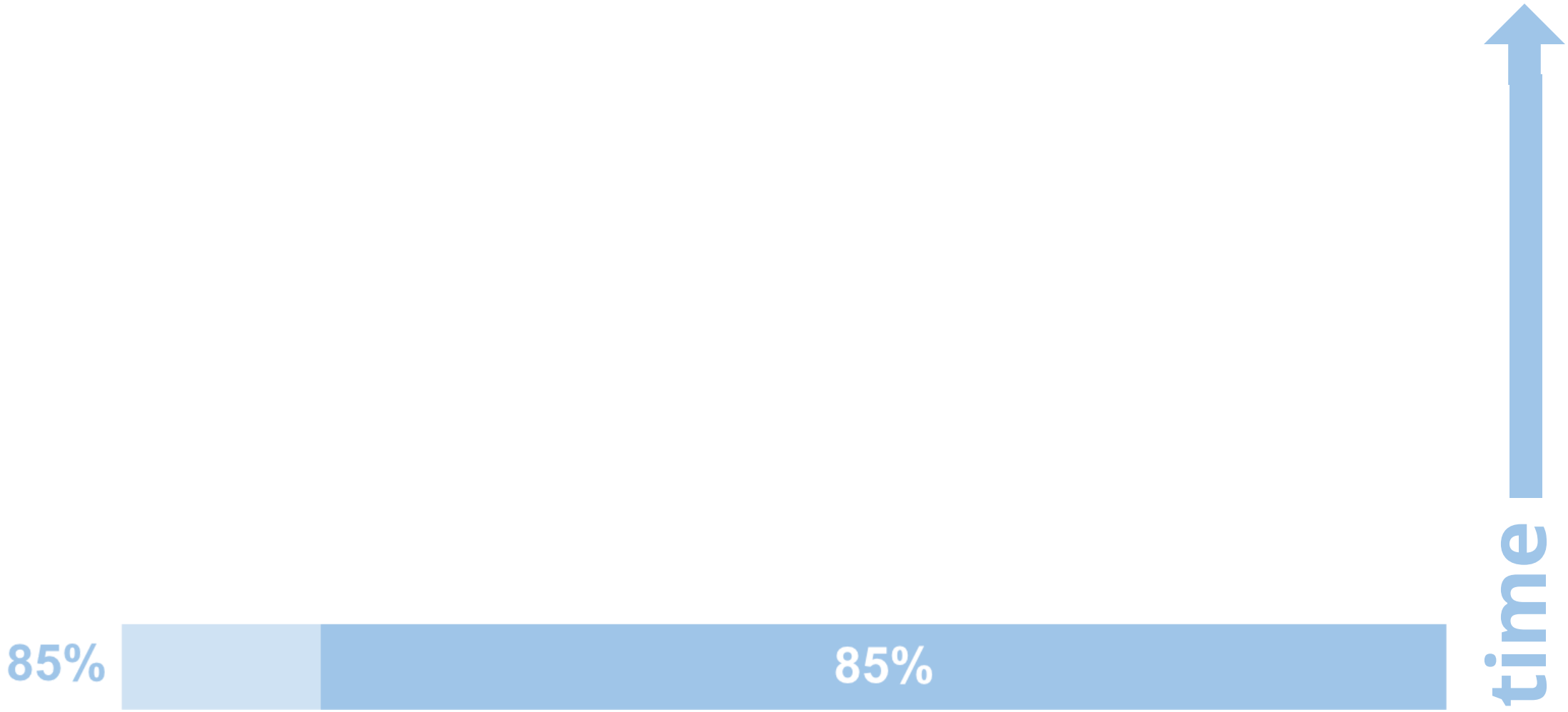


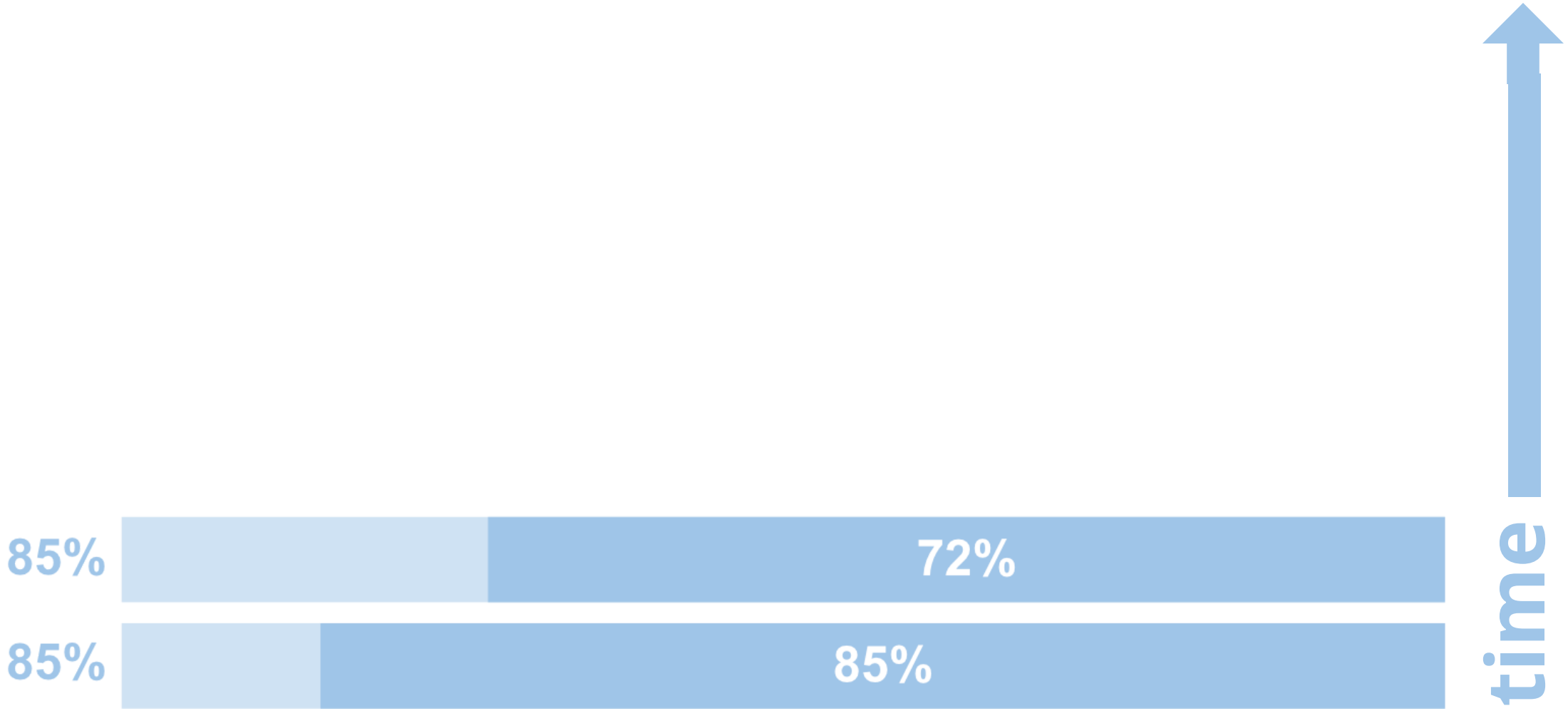

Space of
Possible
Causes

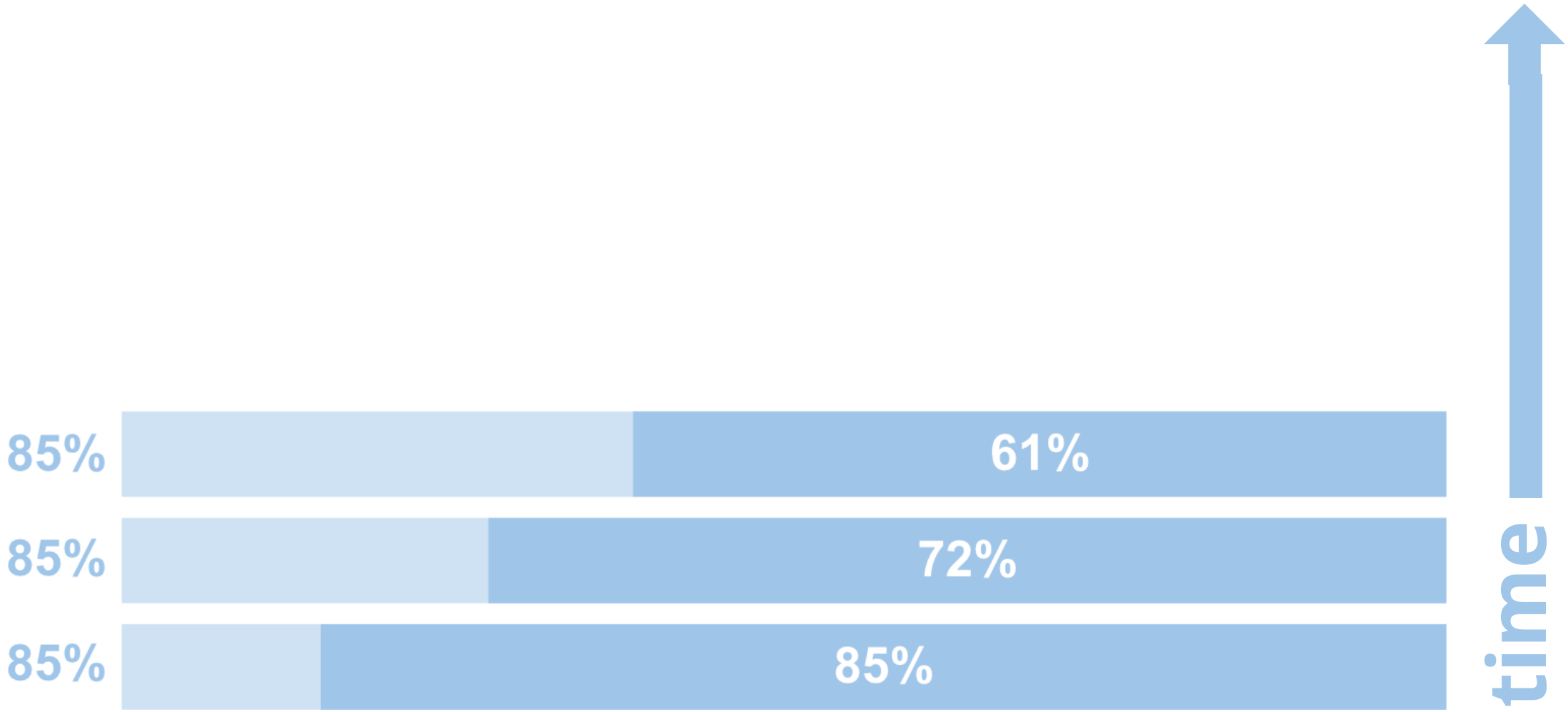

Space of
Possible
Causes

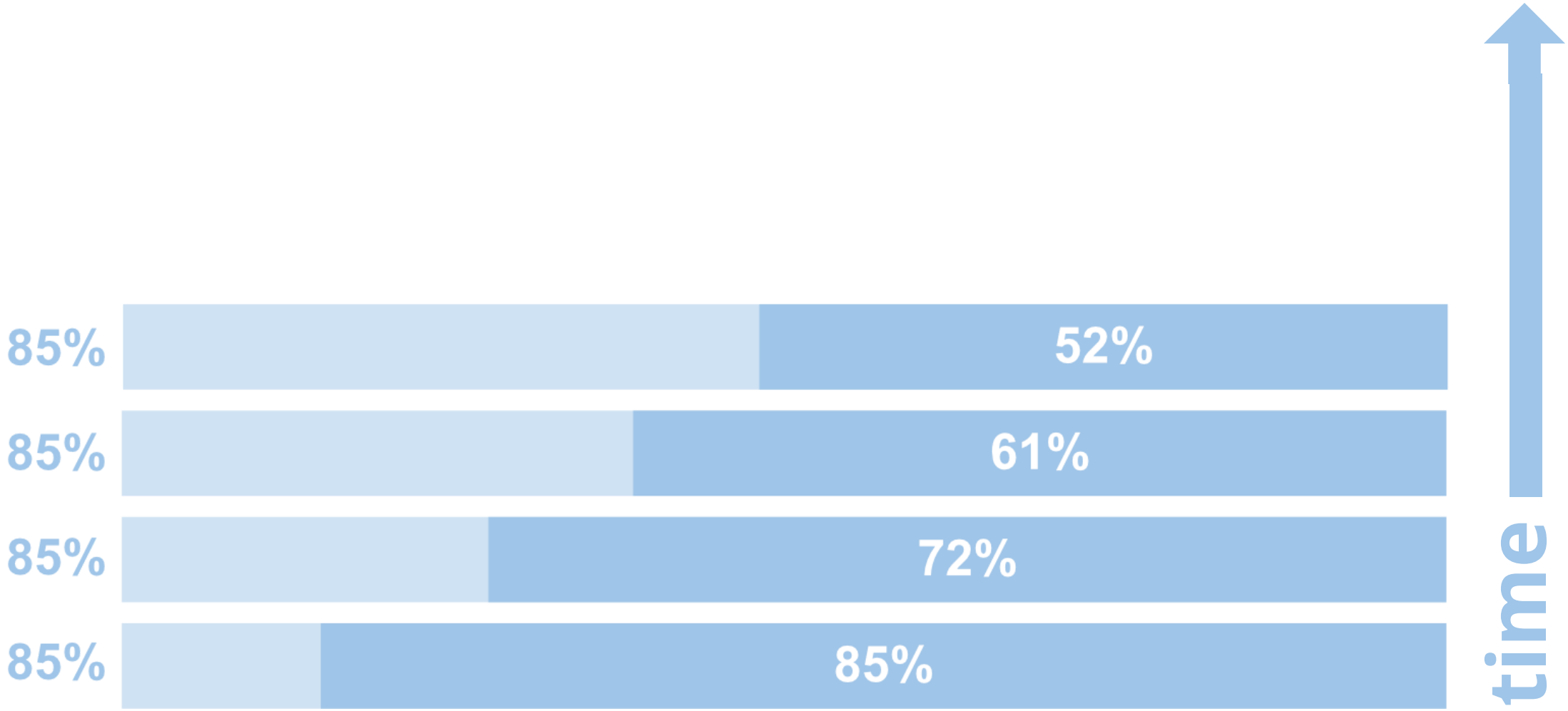

Space of
Possible
Causes

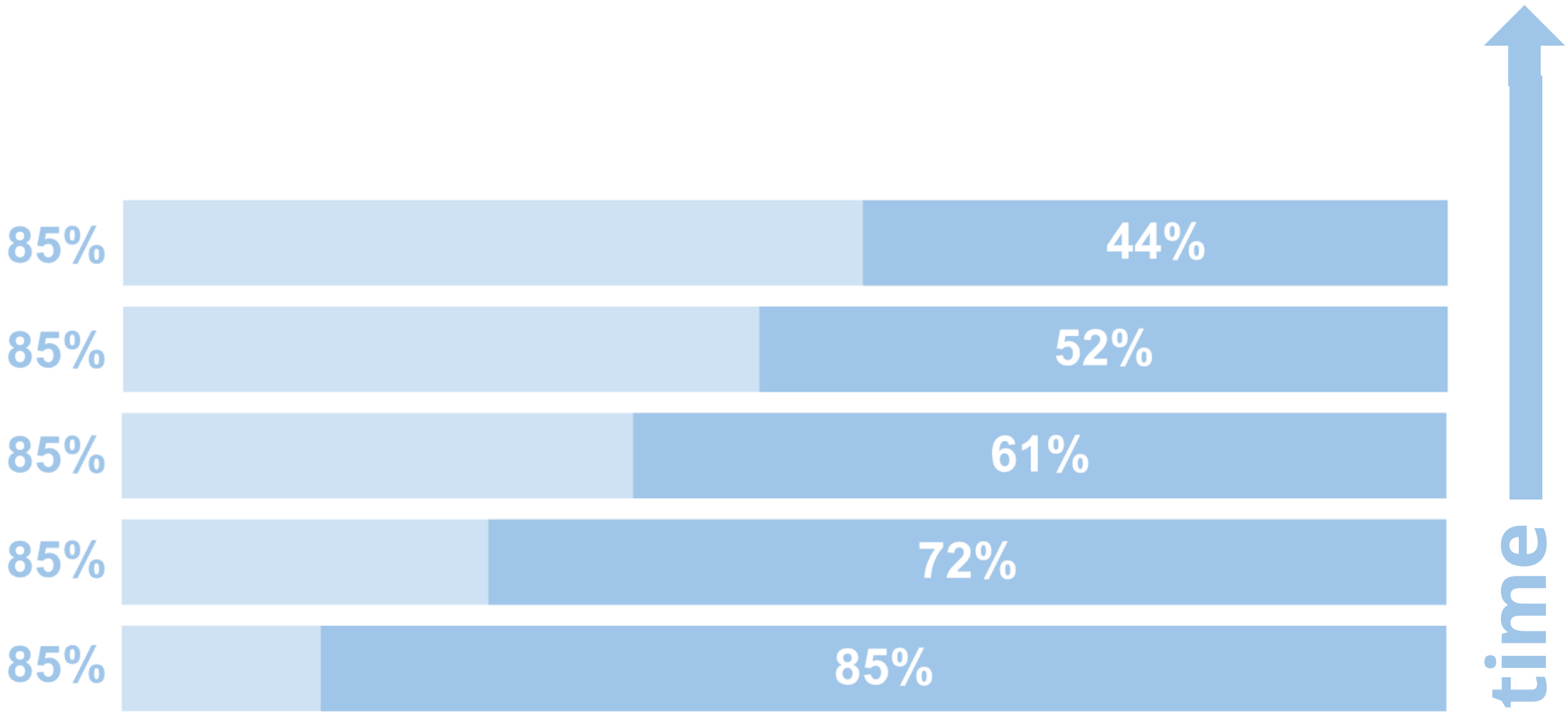
It is **Critical** that you are confident in your **Answers**

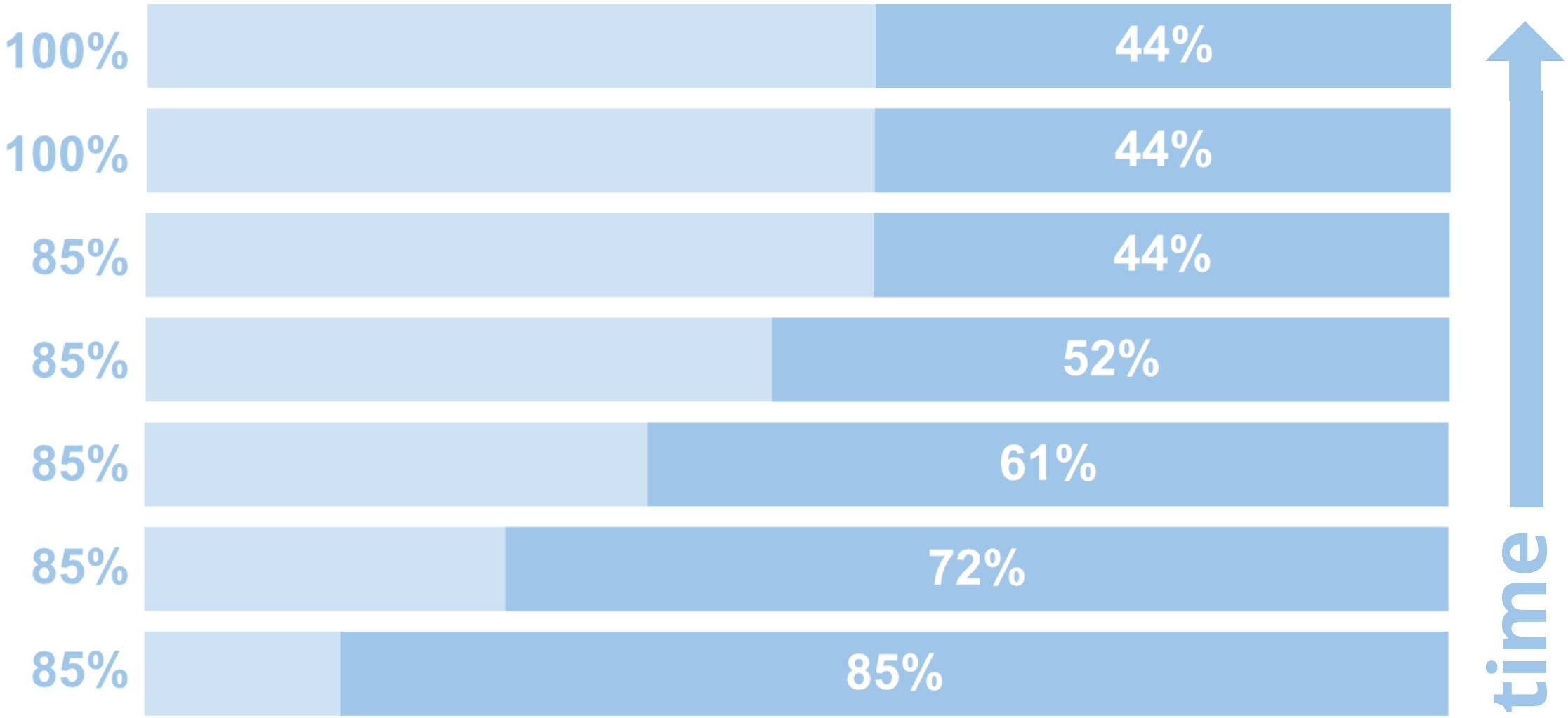

85%
85%
time

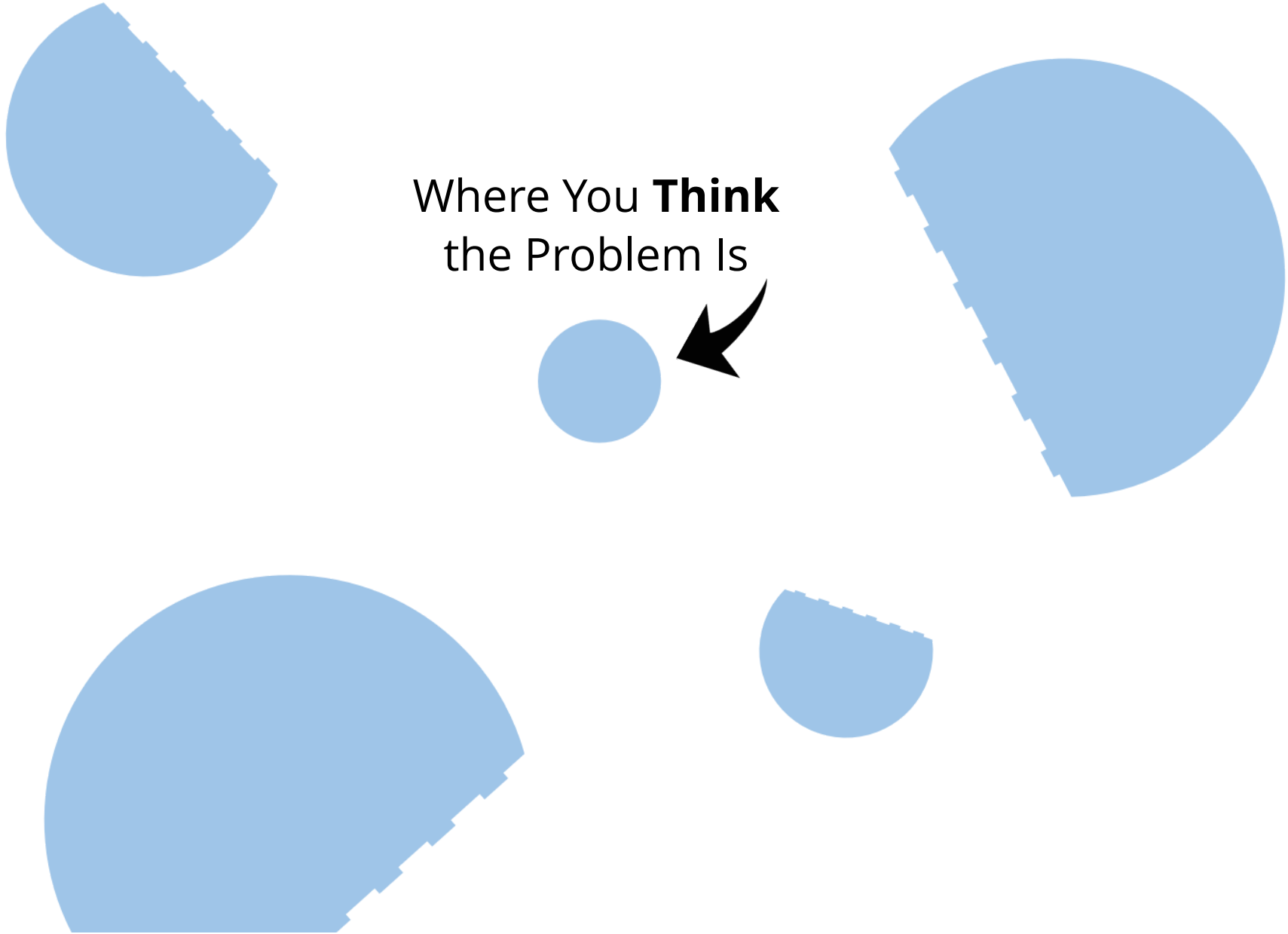

85%
72%
85%
85%
time

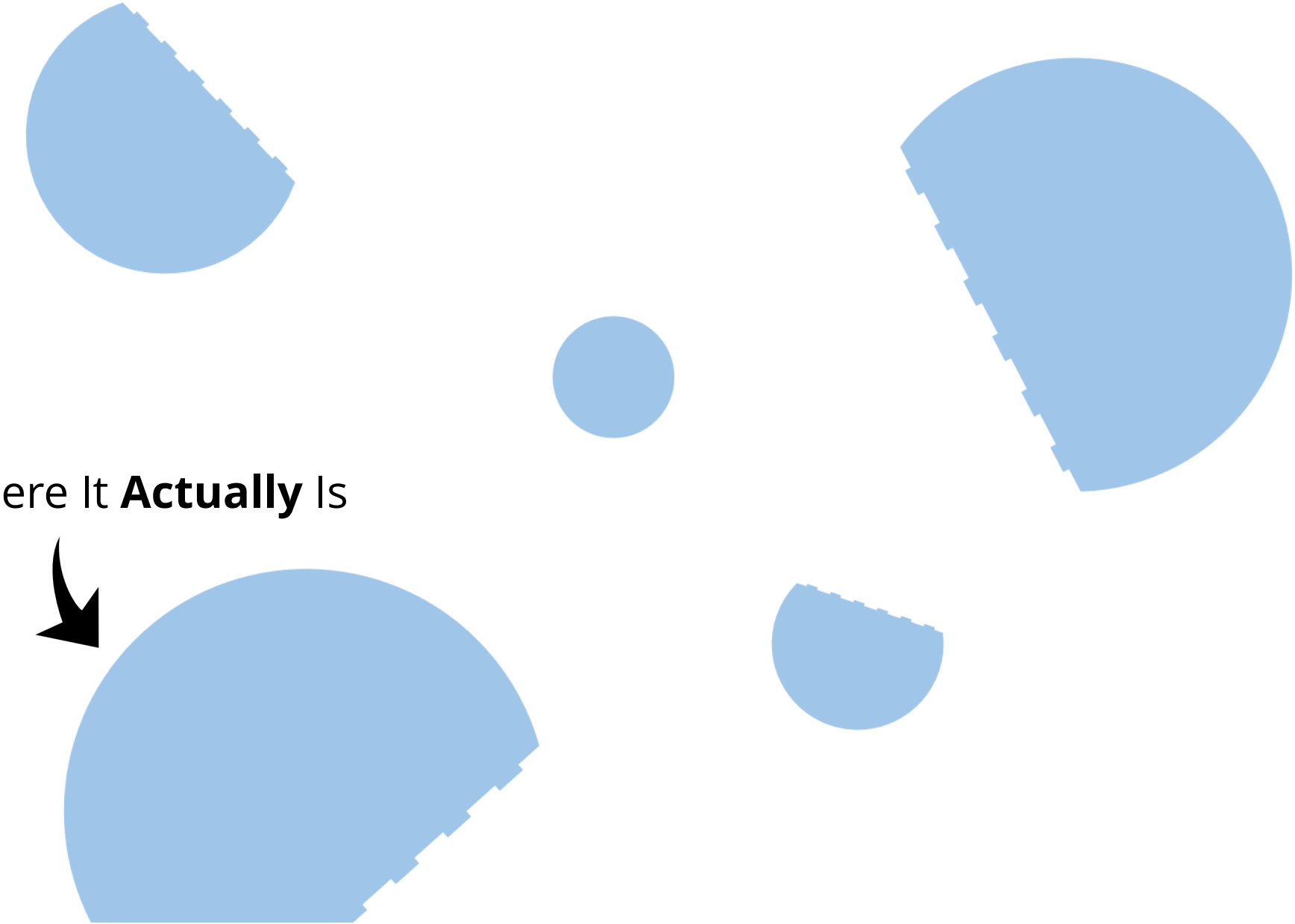

85%
61%
85%
72%
85%
85%
time


85%
52%
85%
61%
85%
72%
85%
85%
time


85%
44%
85%
52%
85%
61%
85%
72%
85%
85%
time


100%
44%
100%
44%
85%
44%
85%
52%
85%
61%
85%
72%
85%
85%
time

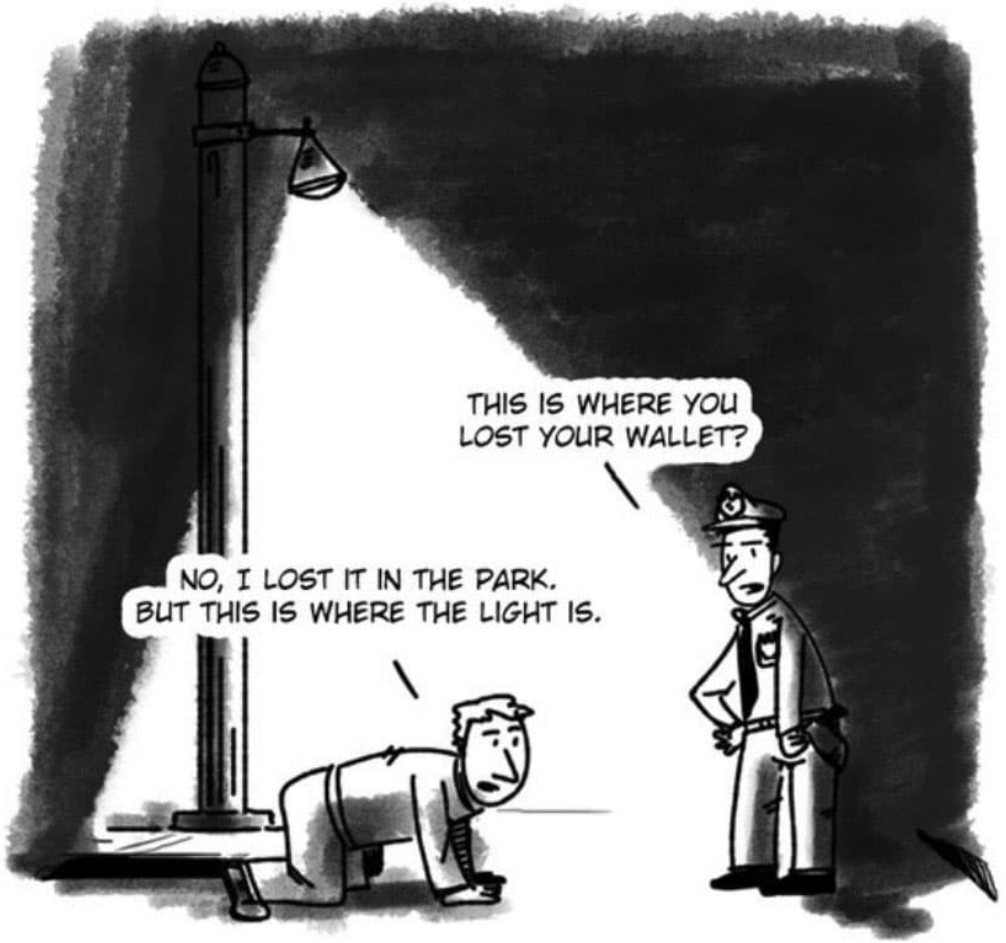

Where You **Think**
the Problem Is


Where It **Actually** Is

Just asking a Question doesn't help
if you can not get the Answer

Not having the right tools constrains the
sort of Questions you can ask

Streetlight Effect



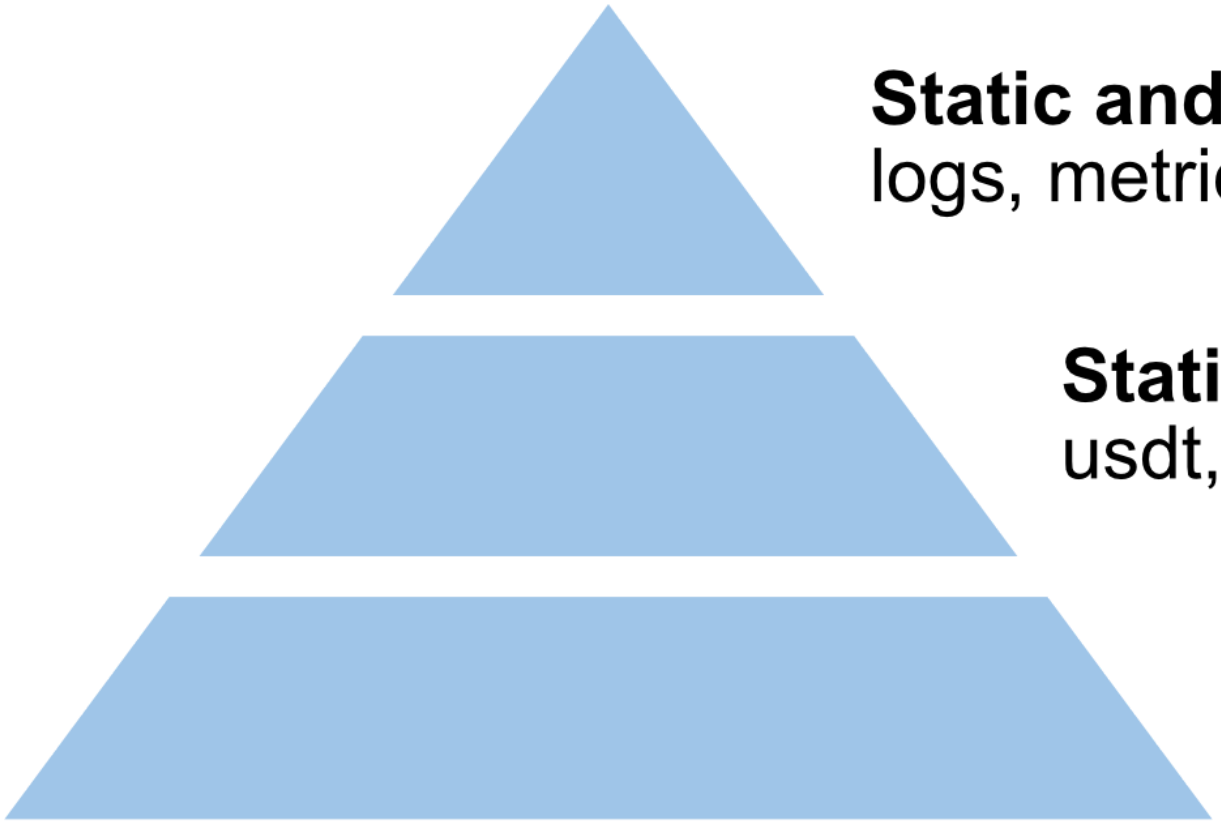
We need Tools that can give us
Answers to our Questions

Method

Tools

Practice

Hierarchy of Instrumentation



Measurement Coverage

Static and Always On
logs, metrics, distributed tracing

Static and Requires Activation
usdt, tracepoints

Dynamic Instrumentation
ptrace, uprobes, kprobes

We want tools that can answer arbitrary questions about our software

Intercept any point of execution

Without restarting the process

Read from memory and registers

Collect data across multiple processes and the kernel

With low overhead

And do it all safely

Debuggers are Awesome

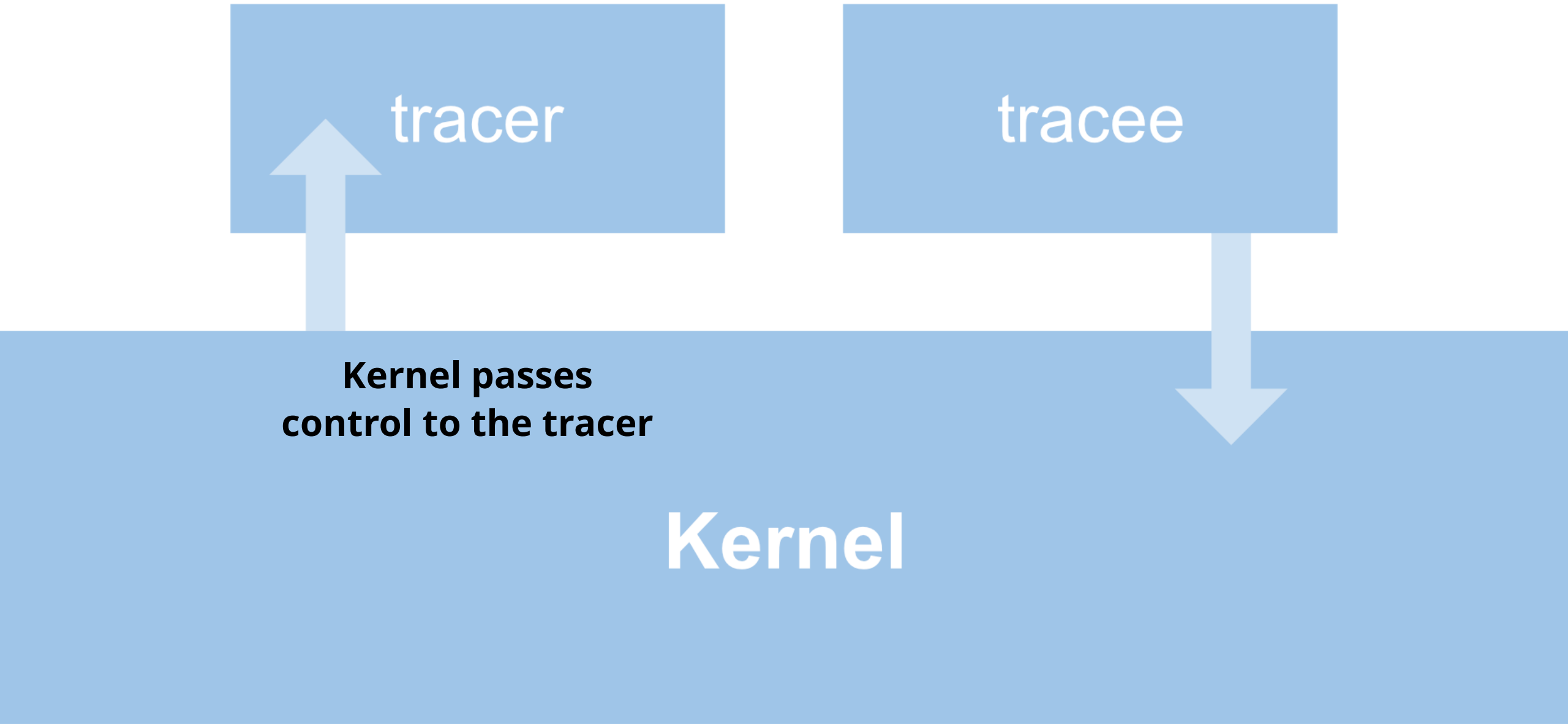
Starting GDB

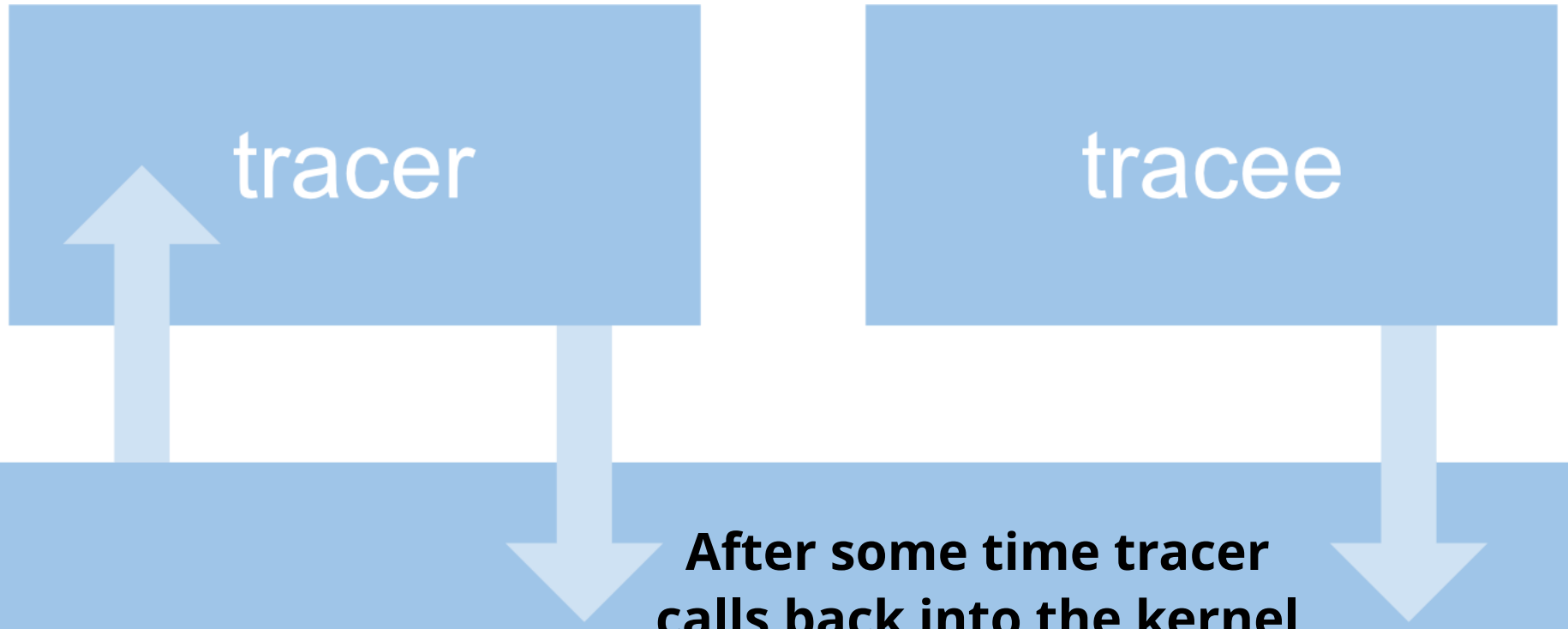
ptrace (Process Trace)

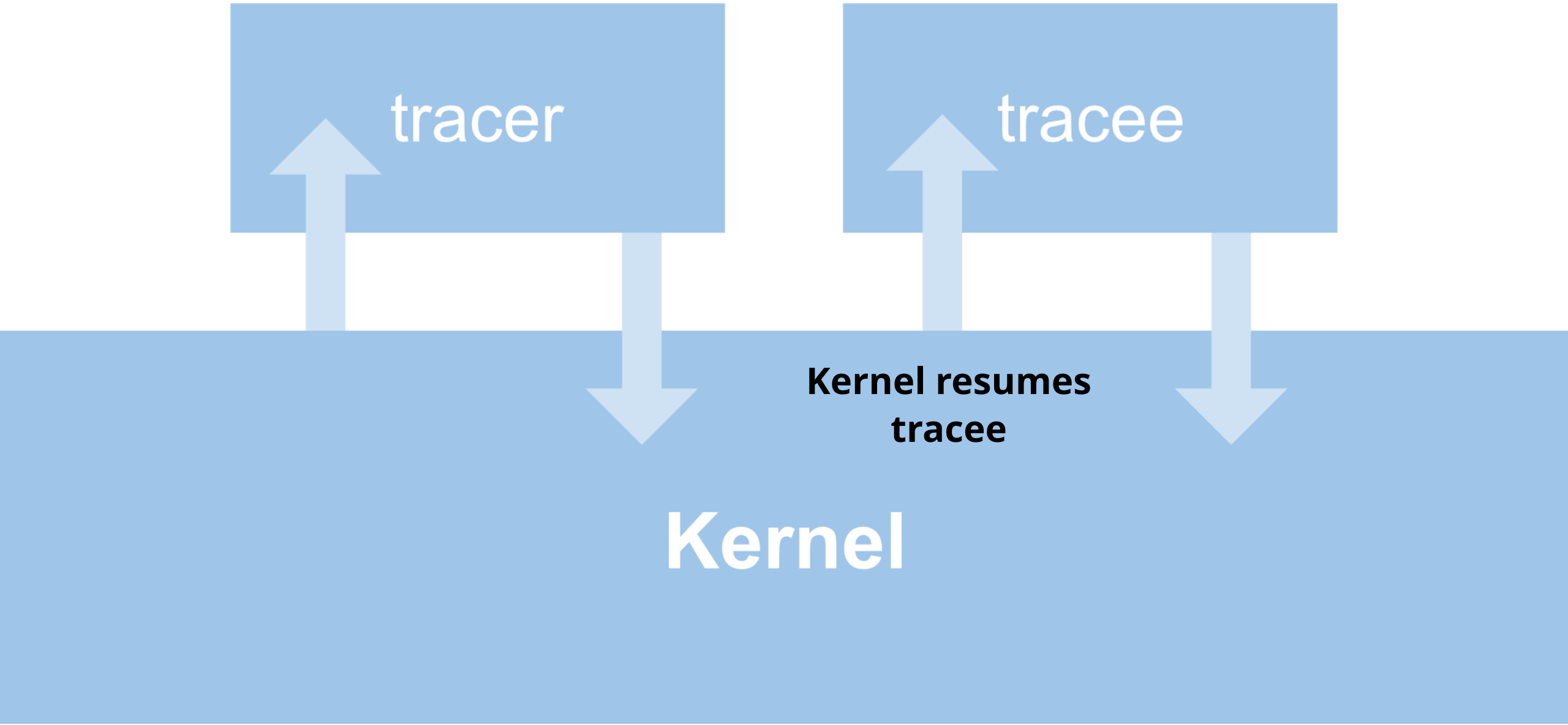
- Allows a tracer process to control the execution of a tracee process
  - intercept signals
  - intercept syscalls
  - read and write to registers/memory (including .text)
  - single step through the tracee
- Writing to .text allows you to set breakpoints
- When tracer is running the tracee's execution is typically suspended

Kernel


tracer
tracee
Trap is Hit
tracee is
suspended
Kernel


tracer
tracee
Kernel passes
control to the tracer
Kernel


tracer
tracee
After some time tracer
calls back into the kernel
Kernel


tracer
tracee
Kernel resumes tracee
Kernel

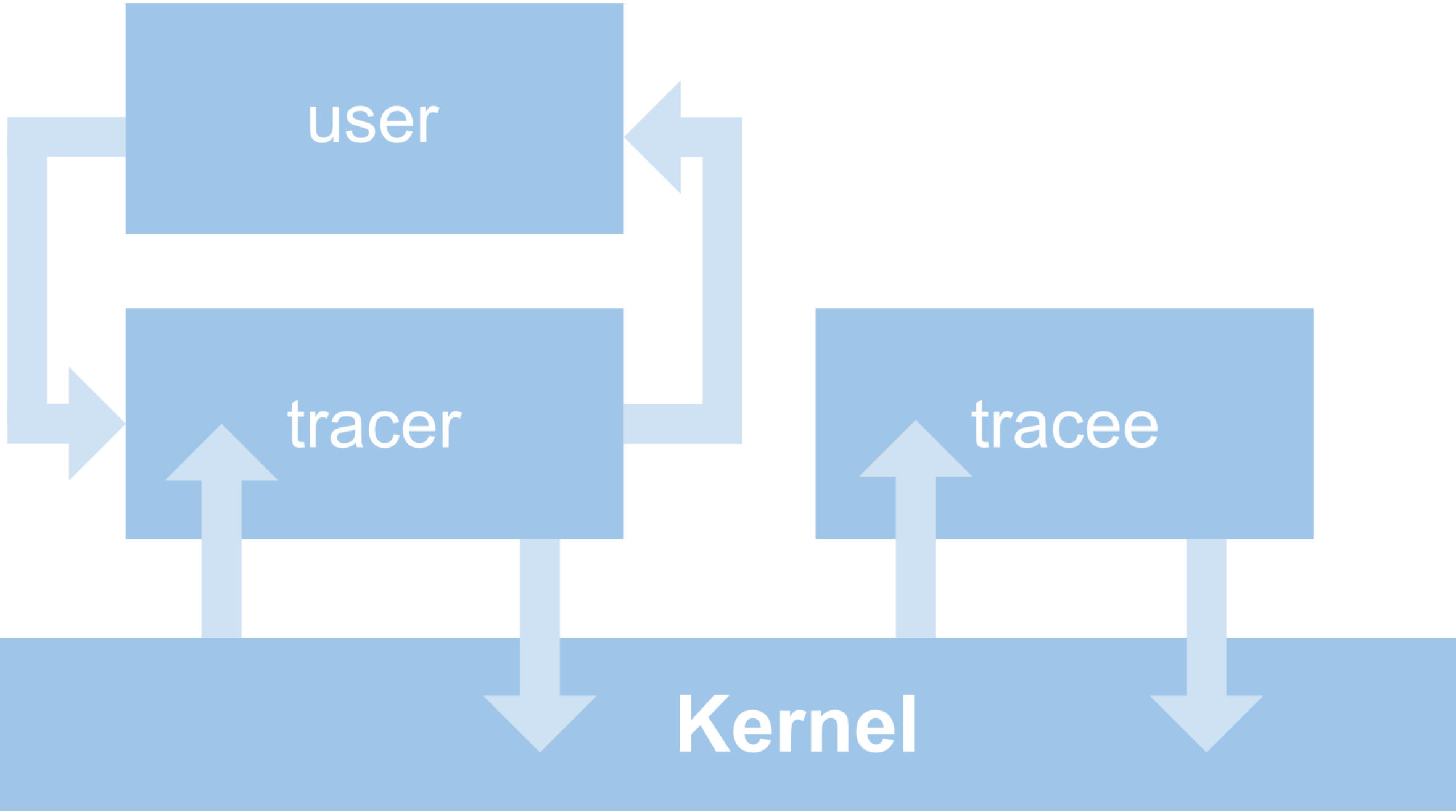
Main Problem with ptrace:

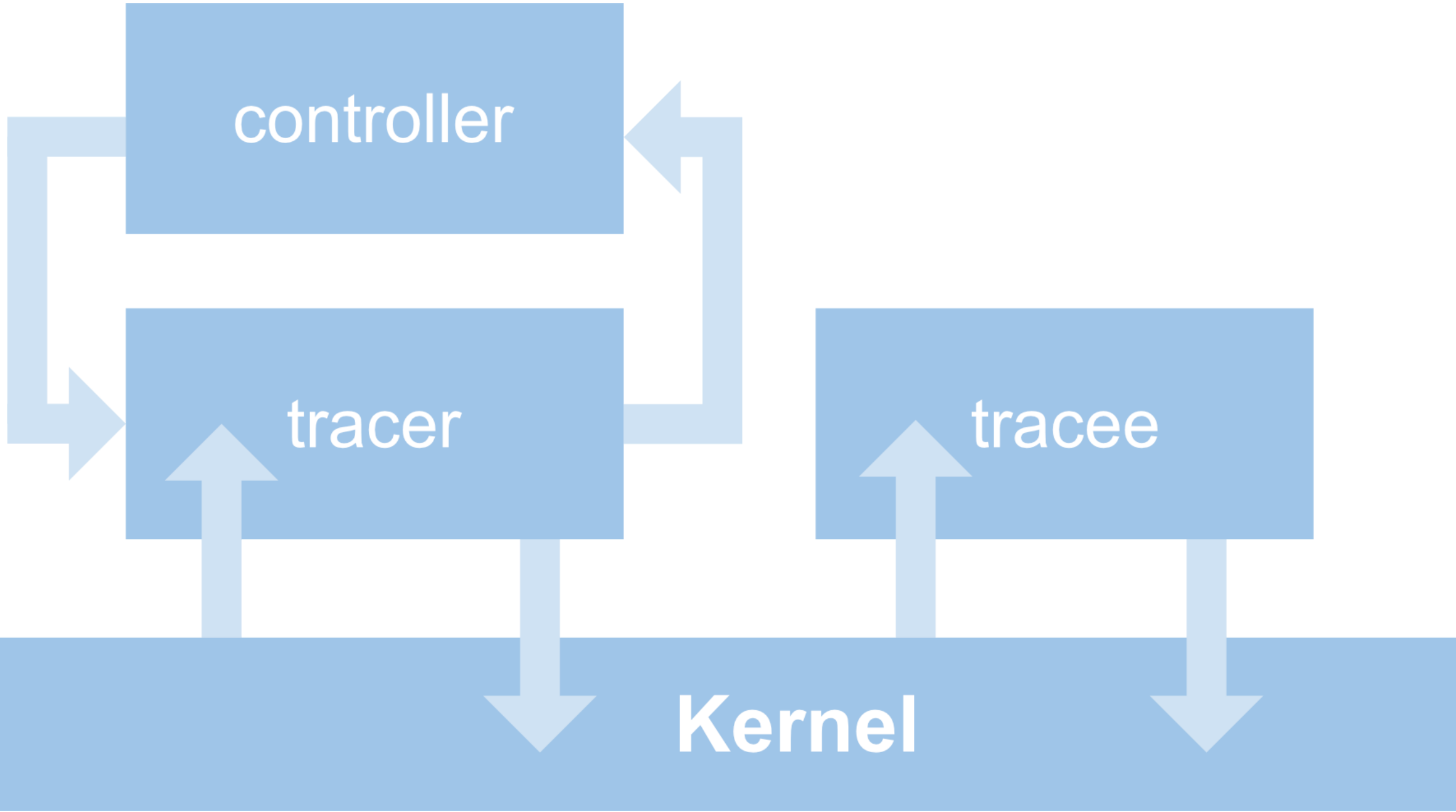
Suspended Execution

Your program is doing **no** work while it is suspended!

If the tracer is slow to yield back this will cripple a process.

The tracer is usually blocked on **user** input.


user
tracer
tracee
Kernel

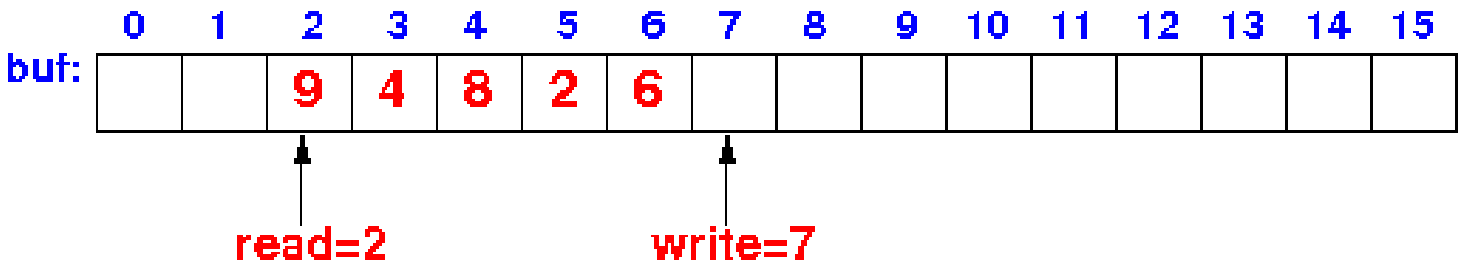

controller
tracer
tracee
Kernel

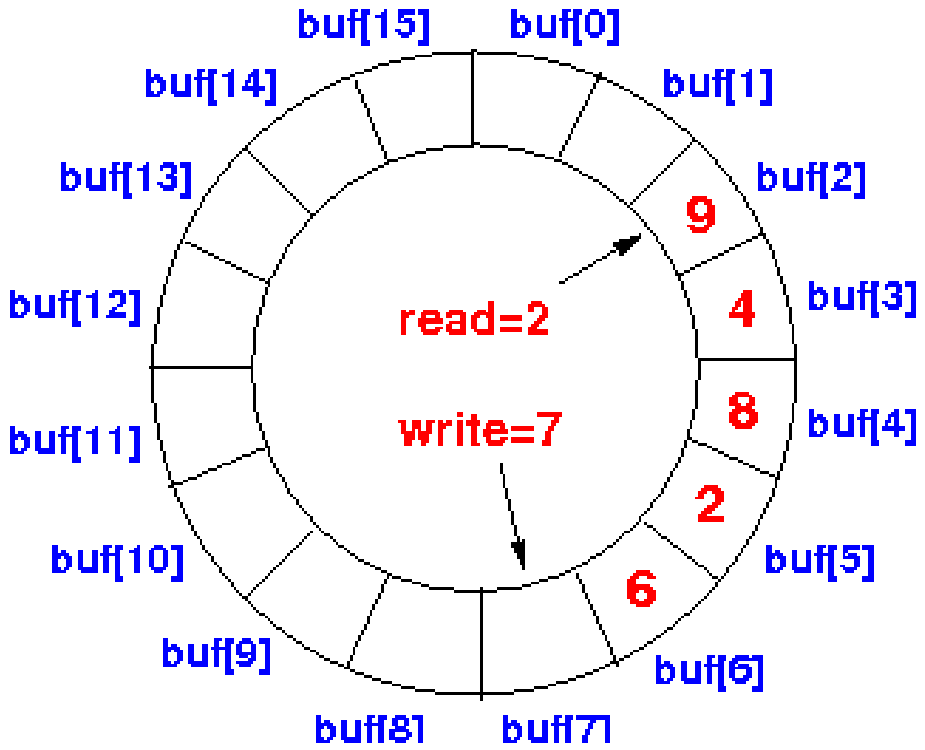
logging agent bug


agent


agent


agent


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
buf:
9 4 8 2 6
read=2
write=7


buf[15]
buf[0]
buf[1]
buf[2]
buf[3]
buf[4]
buf[5]
buf[6]
buf[7]
buf[8]
buf[9]
buf[10]
buf[11]
buf[12]
buf[13]
buf[14]
9
4
8
2
6
read=2
write=7

Question:

When the agent is not sending data, what is the state of the read and write indices?

```
func main() {
        d := diodes.NewOneToOne(1<<12, diodes.AlertFunc(func(int) {}))

        go func() {
                for {
                        write(d)
                }
        }()
        for {
                read(d)
        }
}

func write(d *diodes.OneToOne) {
        d.Set(genData)
}

func read(d *diodes.OneToOne) {
        d.TryNext()
}
```

```
func init() {
        cmd = exec.Command("dlv", "attach", os.Getenv("PID"))
        childIn, _ = cmd.StdinPipe()
        childOut, _ = cmd.StdoutPipe()
}

func main() {
        cmd.Start()

        // resume tracee
        fmt.Fprint(childIn, "continue\n")

        // read, filter and report data
        go reader(childOut)

        for {
                // sample data periodically
        }
}
```

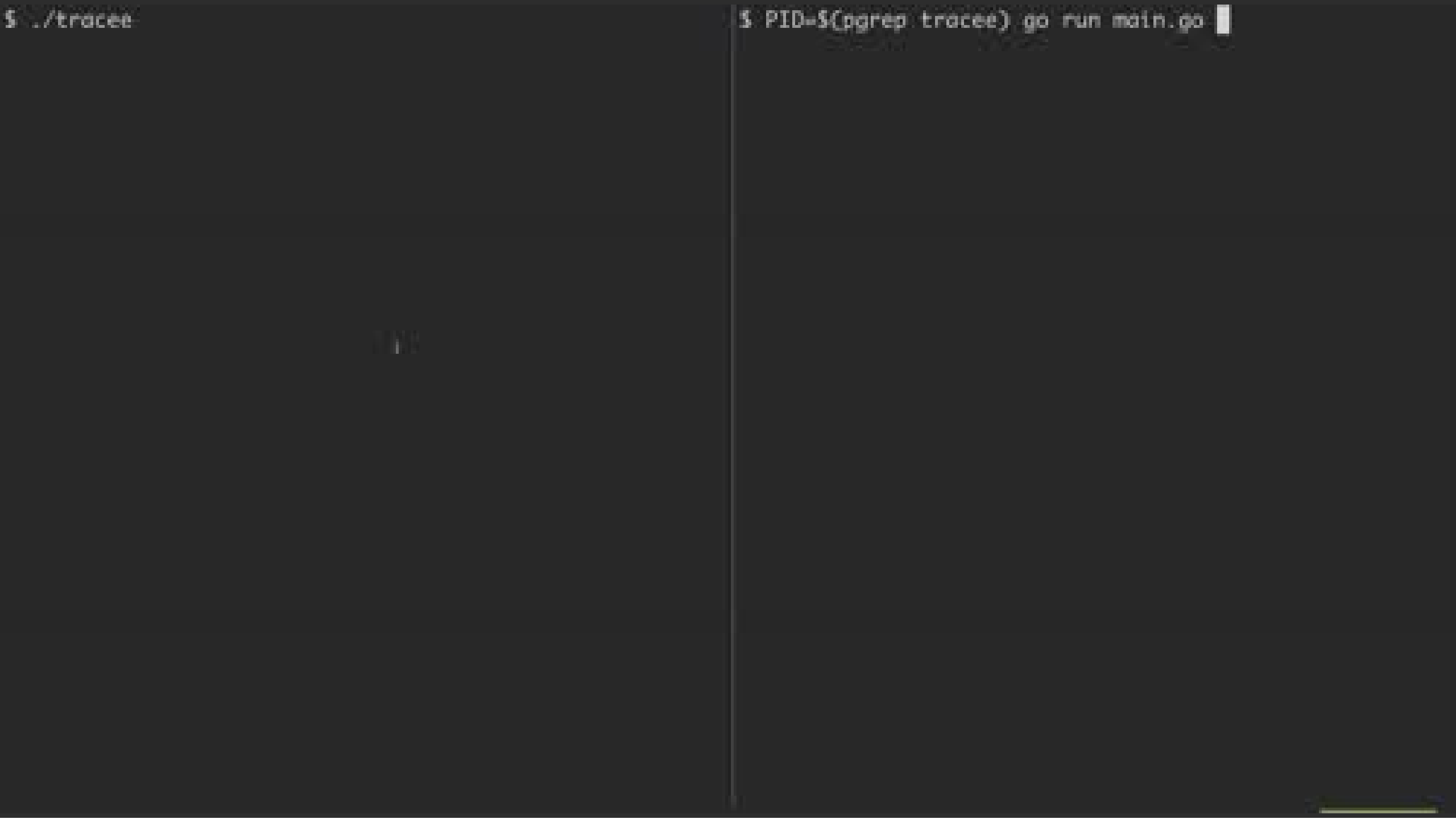
```
time.Sleep(time.Second)
cmd.Process.Signal(os.Interrupt)

if timeToExit() {
        fmt.Fprint(childIn, exit)
        return
}

fmt.Fprint(childIn, sample)
```

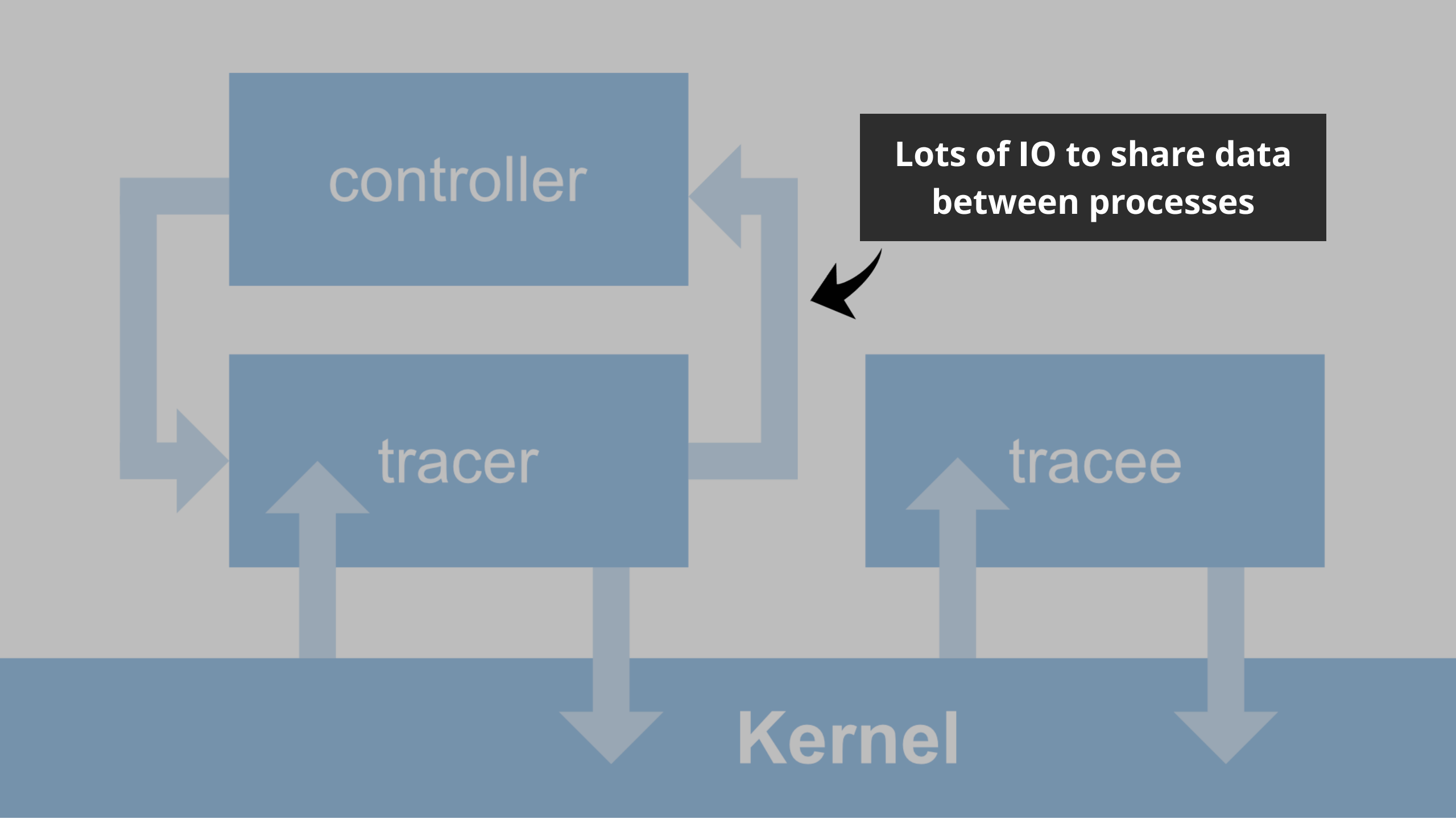
```
const sample = `break main.write
continue
print d.writeIndex - d.readIndex
clearall
continue
`
const exit = `clearall
quit
no
`
```

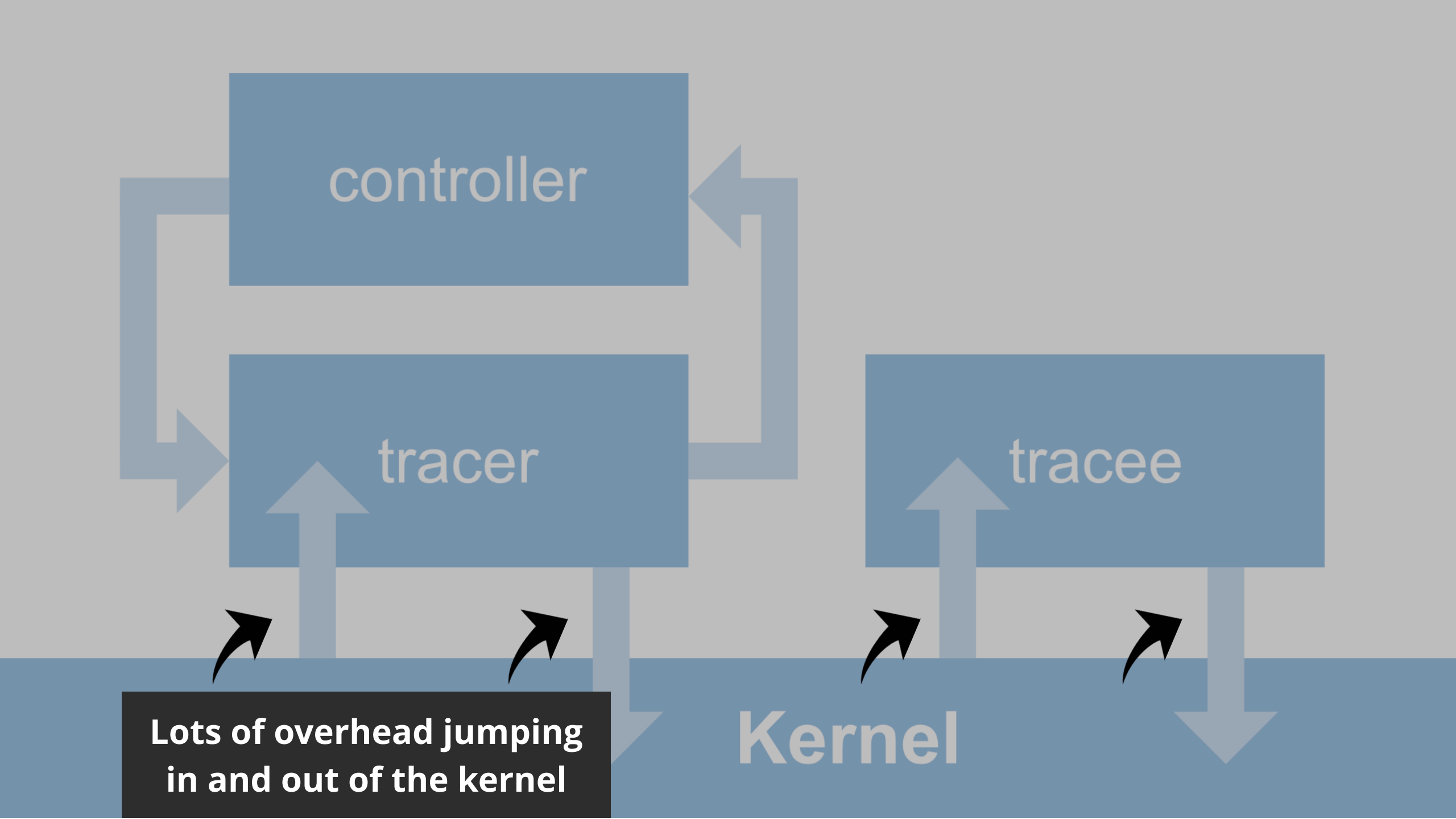
```
$ ./tracee
```

```
$ PID="$(pgrep tracee)" go run main.go
```

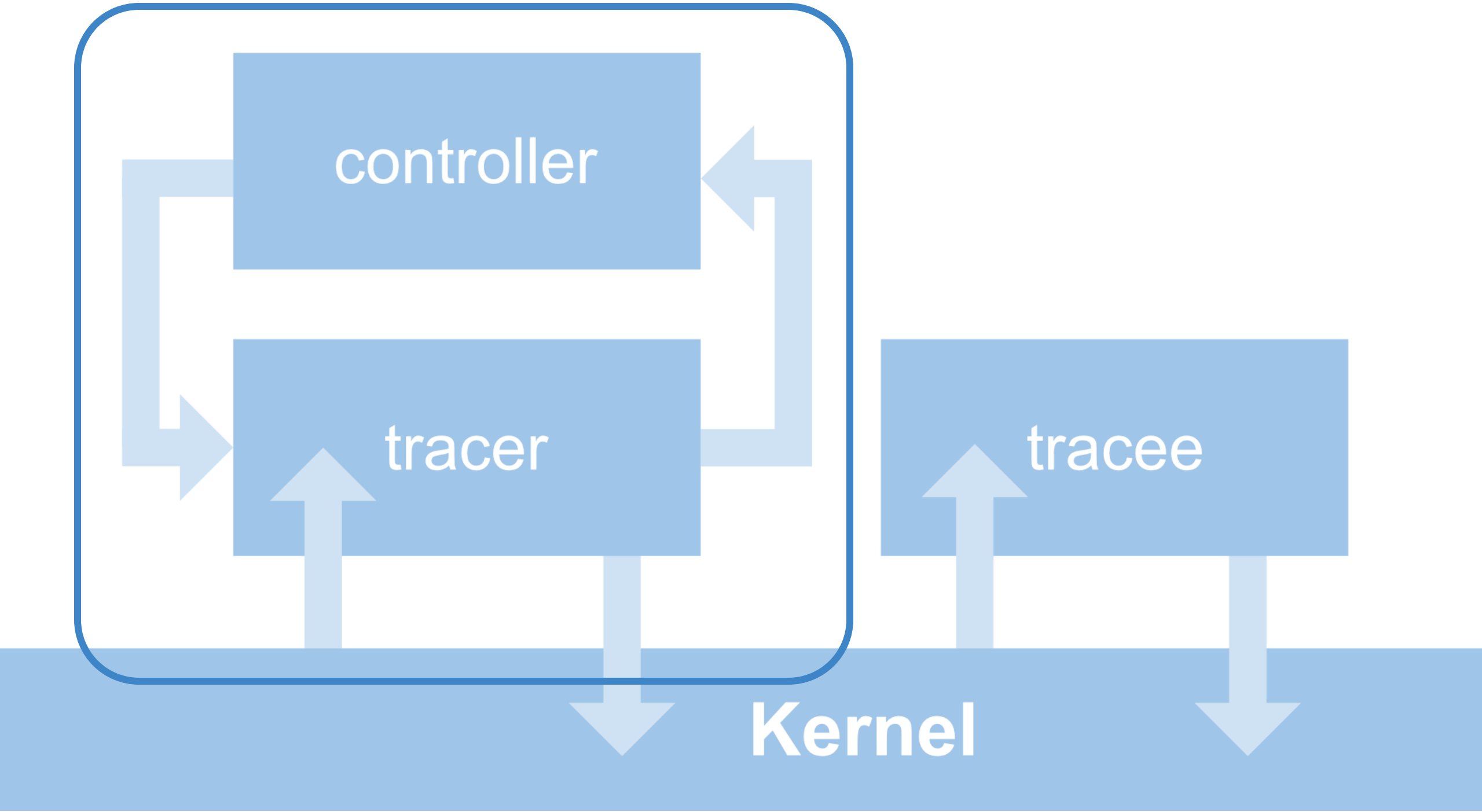
This only works for sampling at a **low frequency**

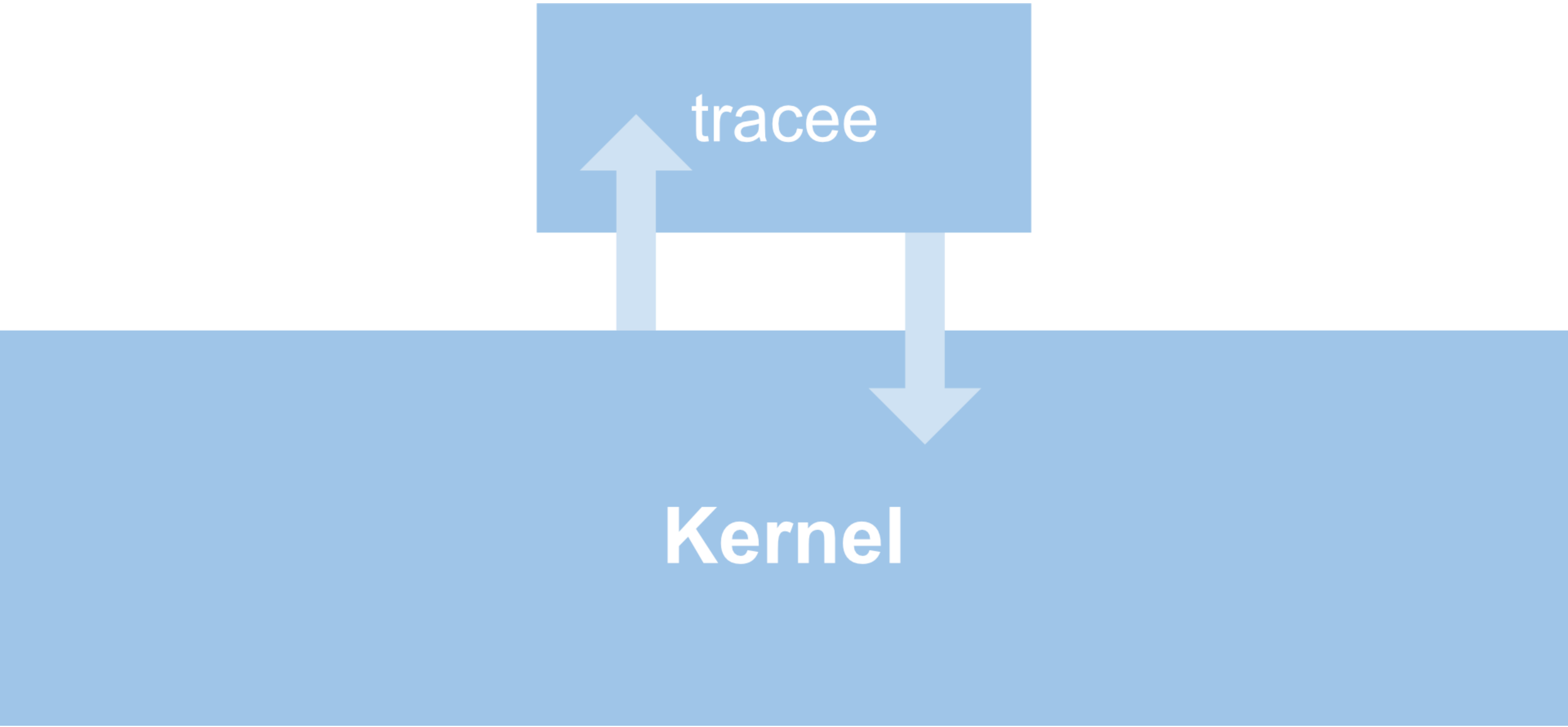

$ ./tracee
$ PID=$(pgrep tracee) go run main.go


controller
tracer
tracee
Kernel
Lots of IO to share data between processes


controller
tracer
tracee
Kernel
Lots of overhead jumping in and out of the kernel

Is There a Better Way?


controller
tracer
tracee
Kernel


tracee
Kernel

BPF can do this!

What is BPF?

- BPF is a custom instruction set that you can use to build programs and inject them into the kernel.
- The kernel validates the program to make sure it is safe and then compiles it for your architecture so it runs fast.
- You can then attach these programs to various events.
- It was originally created for programs that do packet filtering with little overhead, hence the name (Berkeley Packet Filter).
- For example:

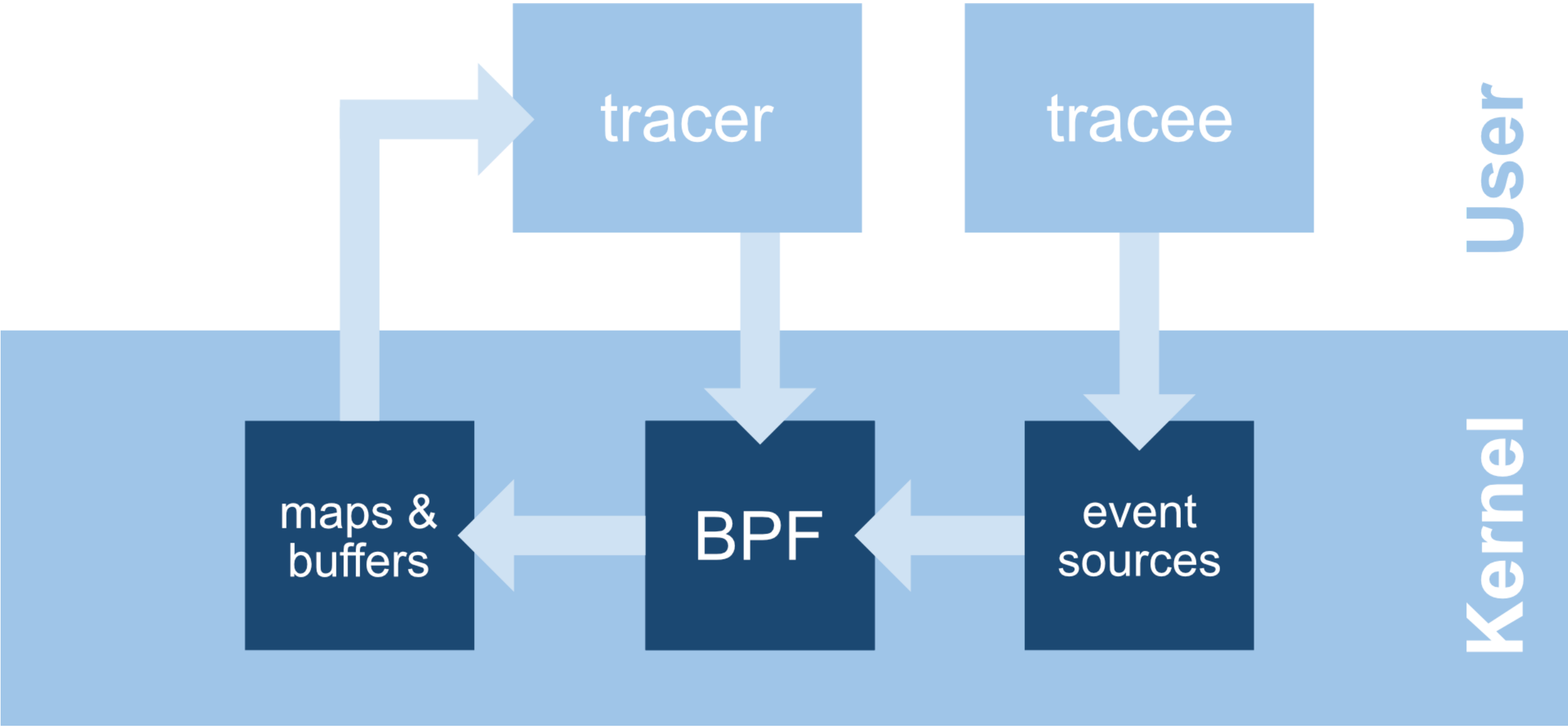
```
tcpdump src 10.5.2.3 and dst port 3389
```

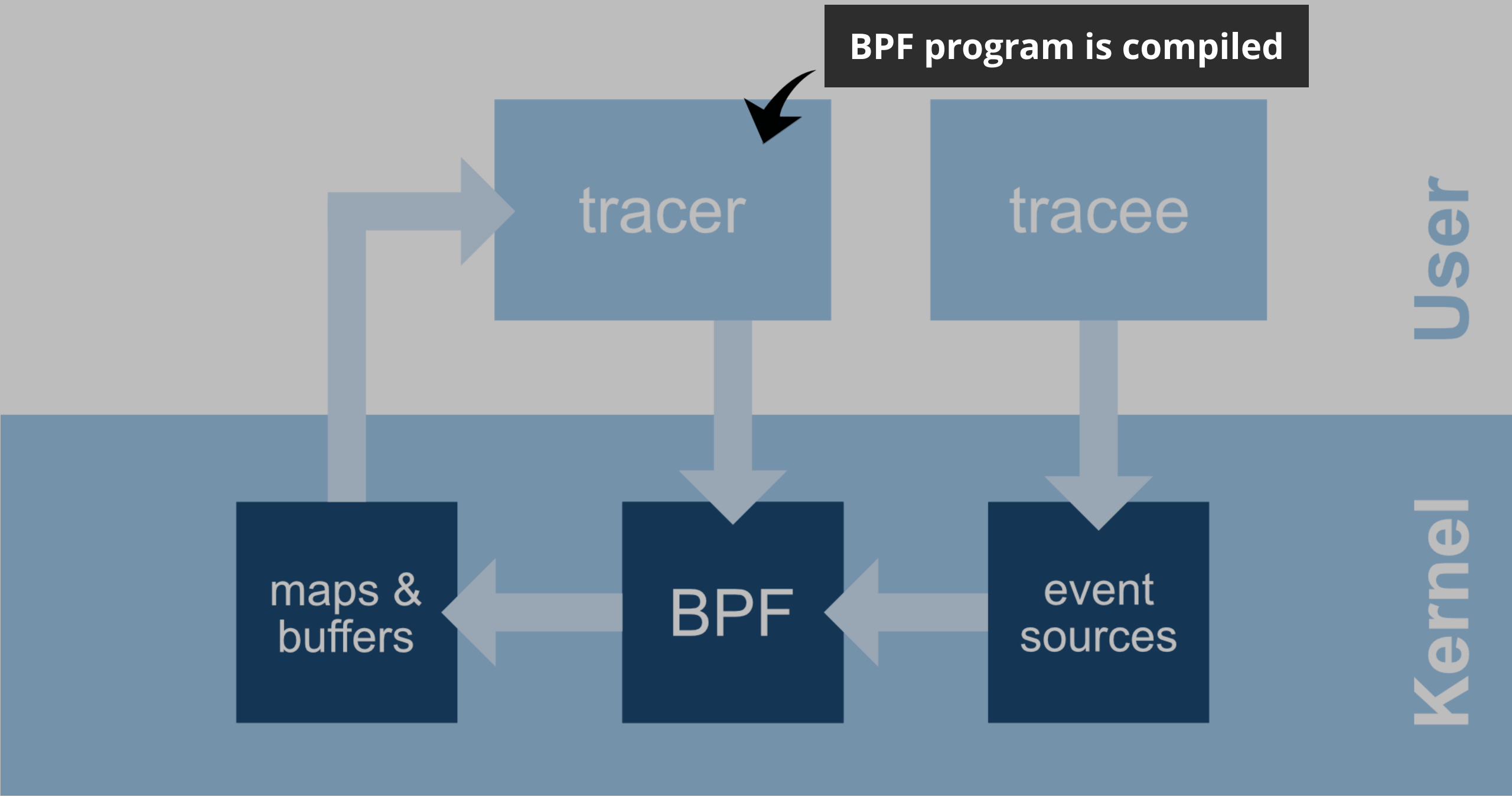
What can your BPF program do?

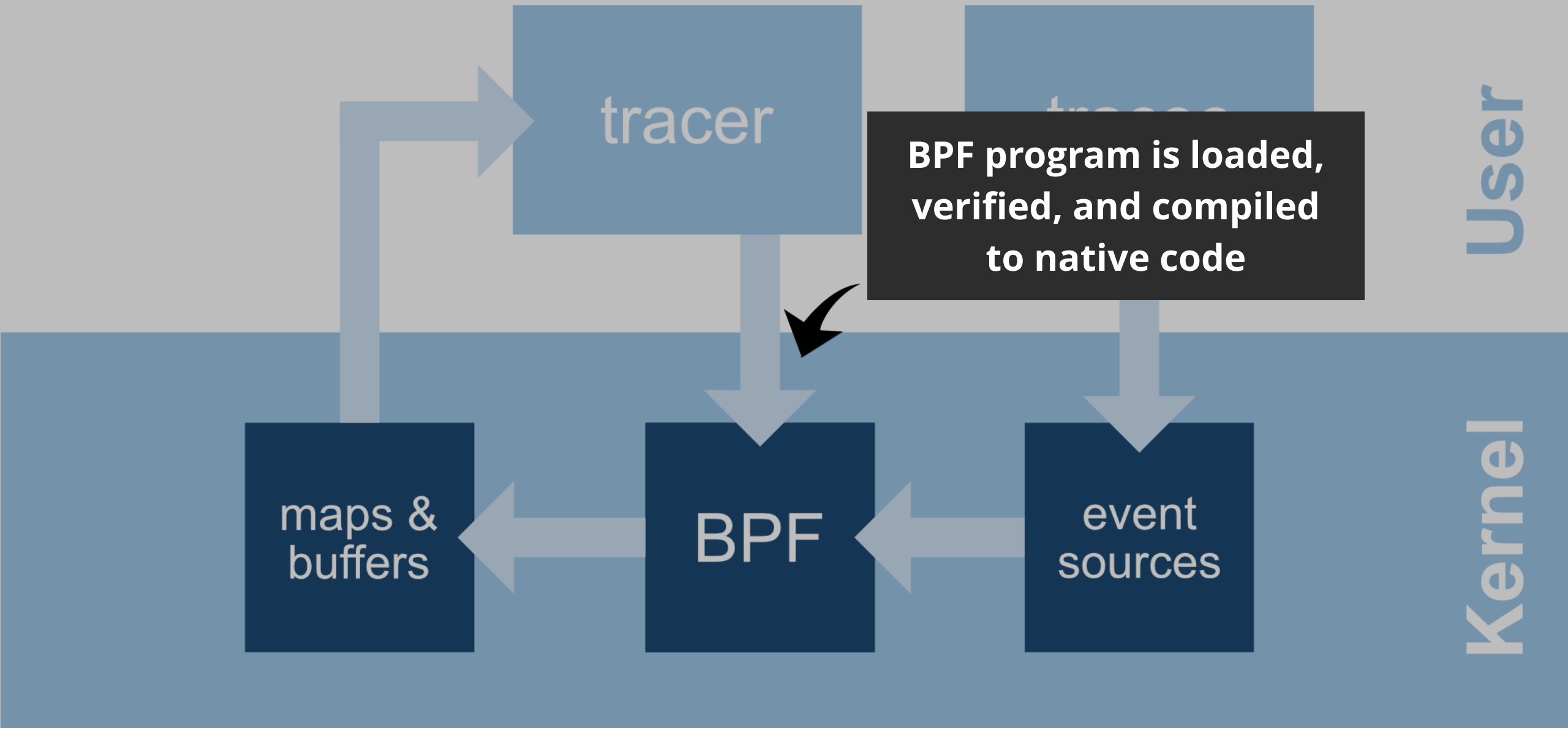
- Arithmetic/Logic/Branching
- Load/Store (restricted)
- Call user defined bpf functions
- Call various helper functions
  - Aggregate and store data in maps
  - Read stack traces for kernel and user land
  - Manipulate packets
  - Get time/rand data/current pid/task/etc
  - Read/write to certain places in memory
  - Much more!

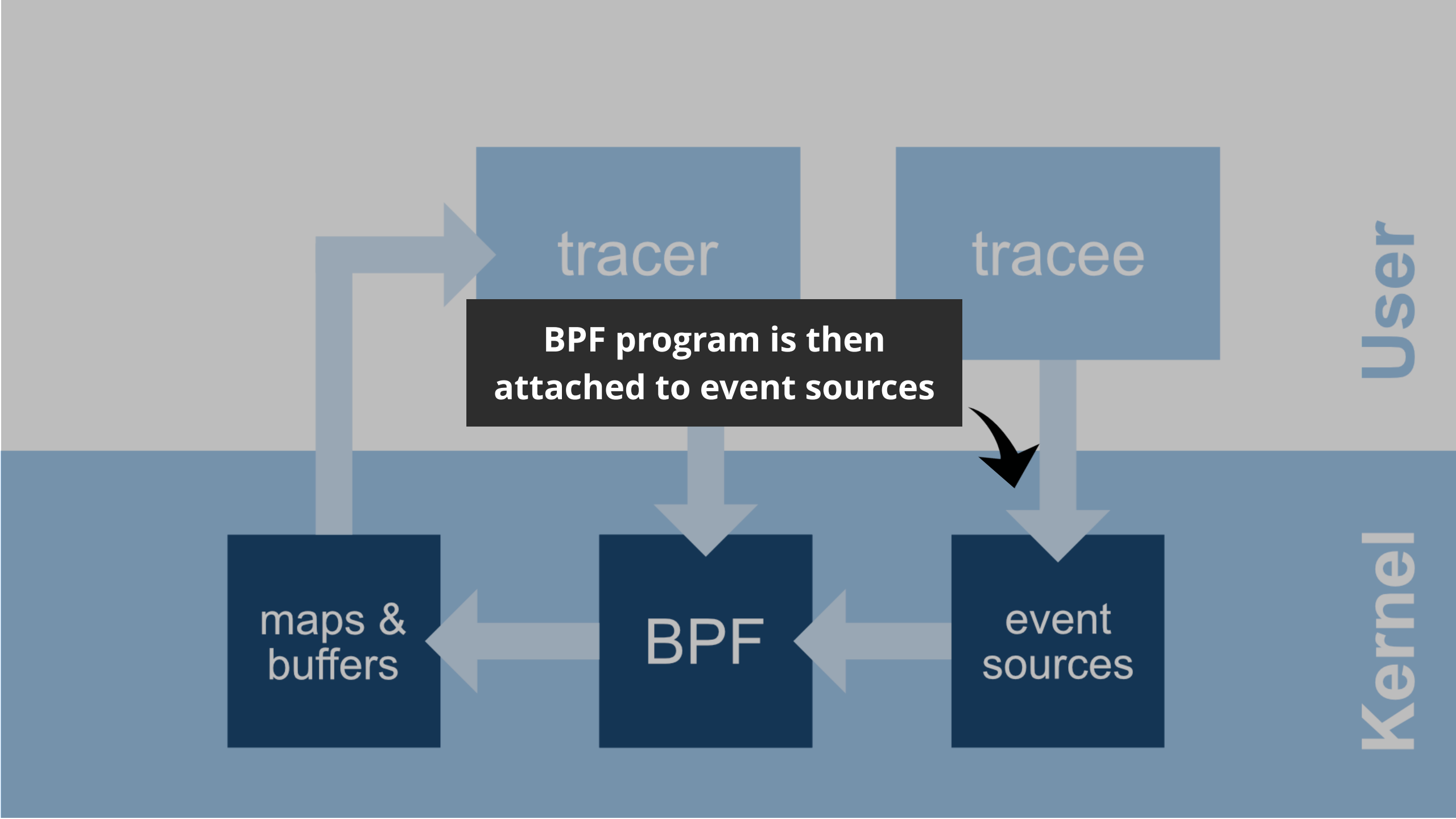
What can your BPF program not do?

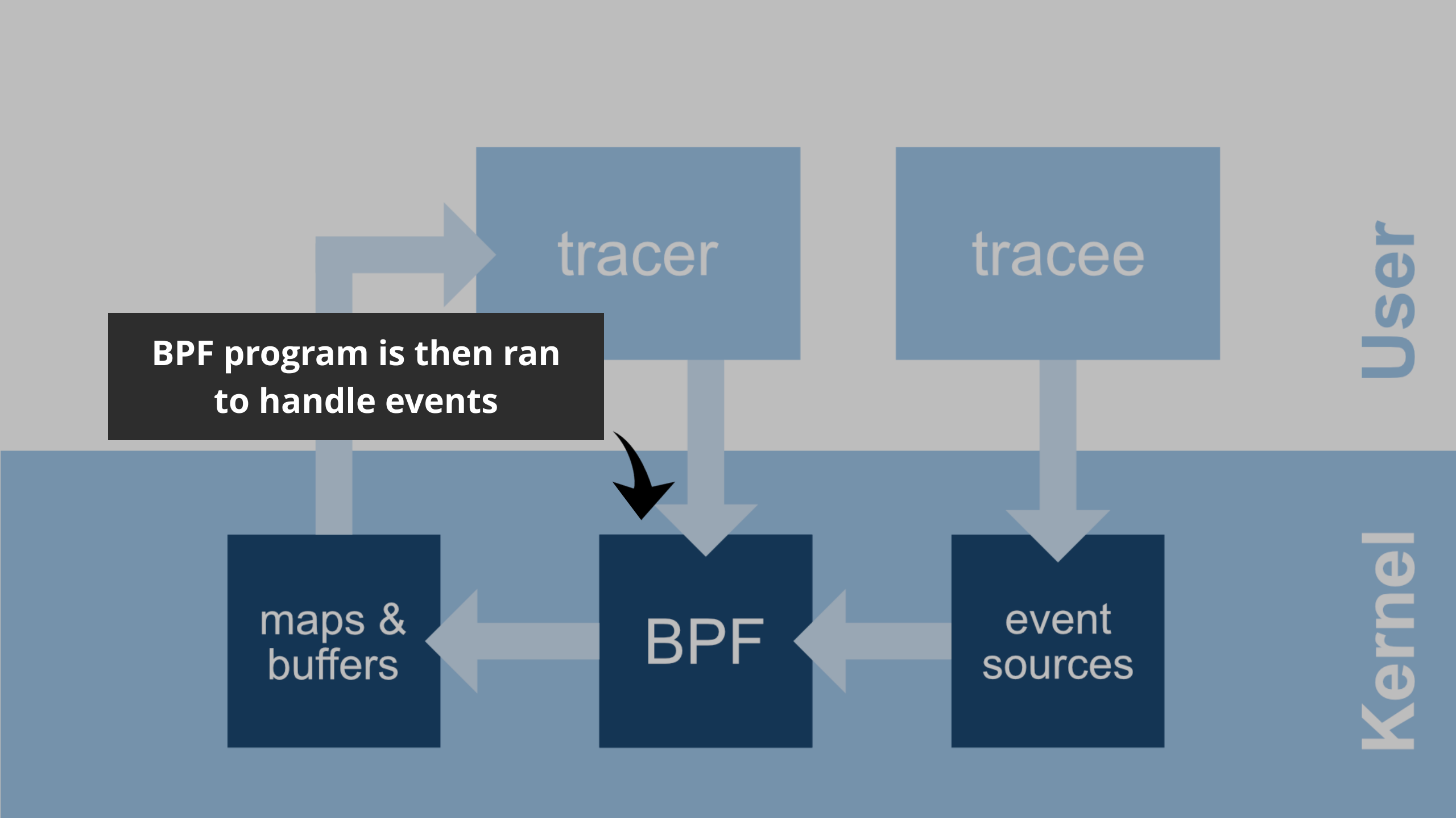
- Your program must have a finite execution
- Loops are not allowed
  - You can jump forward
  - You can jump back if it does not form loop
  - Bounded loops might be allowed in the future so you do not have to manually unroll loops
- Access to locks are not permitted (might be allowed in the future)
- Access to arbitrary memory is not permitted
  - You can load/store the memory of the BPF program and access memory in other ways
- No illegal instructions
- Unreachable blocks are not allowed

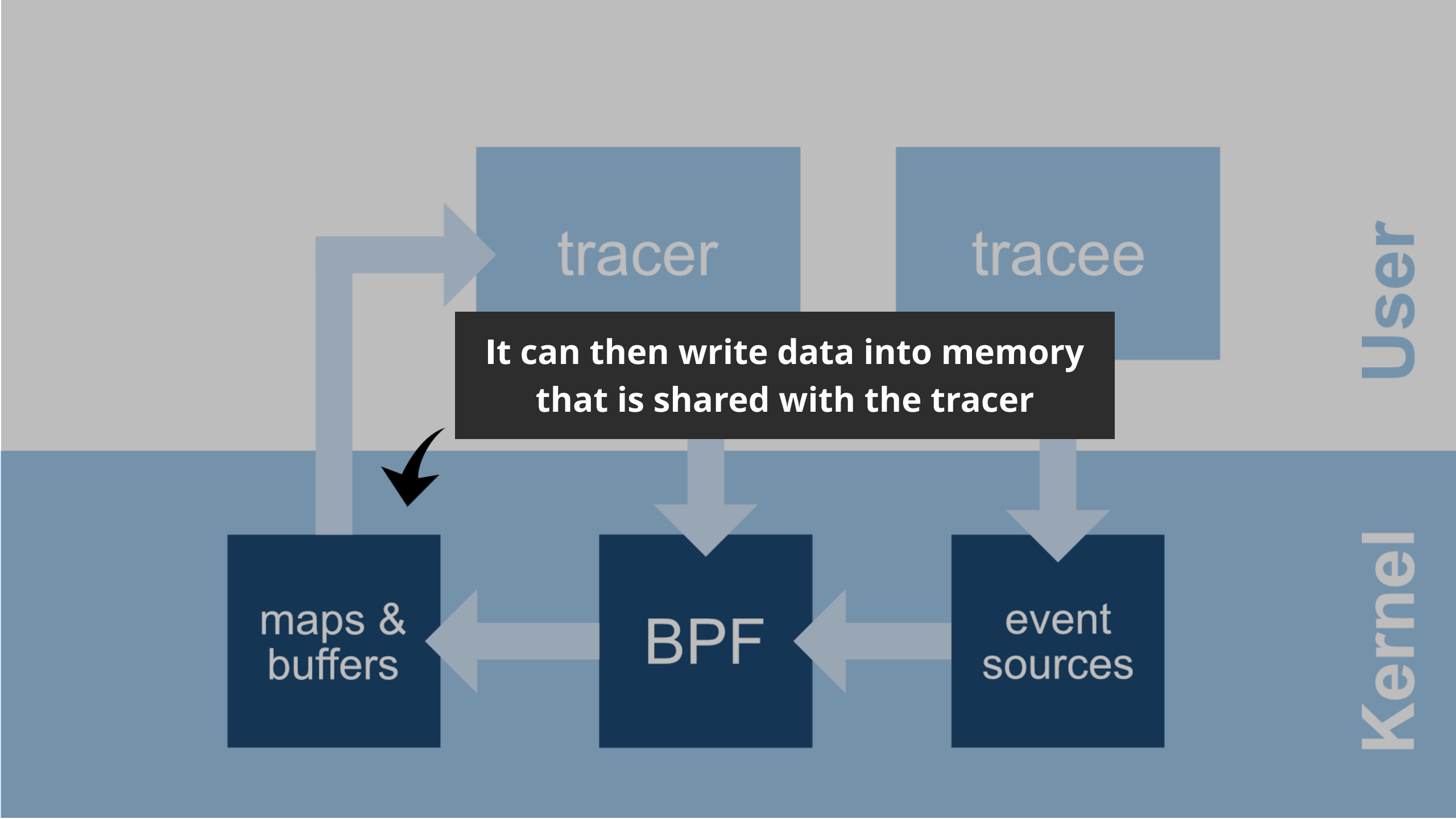

tracer
tracee
User
maps &
buffers
BPF
event
sources
Kernel


BPF program is compiled
tracer
tracee
User
maps &
buffers
BPF
event
sources
Kernel


tracer
BPF program is loaded, verified, and compiled to native code
User
maps & buffers
BPF
event sources
Kernel


tracer
tracee
BPF program is then attached to event sources
maps & buffers
BPF
event sources
User
Kernel


tracer
tracee
User
BPF program is then ran
to handle events
maps &
buffers
BPF
event
sources
Kernel


tracer
tracee
It can then write data into memory
that is shared with the tracer
maps &
buffers
BPF
event
sources
User
Kernel

Event Sources

User Space

- uprobes - dynamic
- usdt - static (uses uprobes)

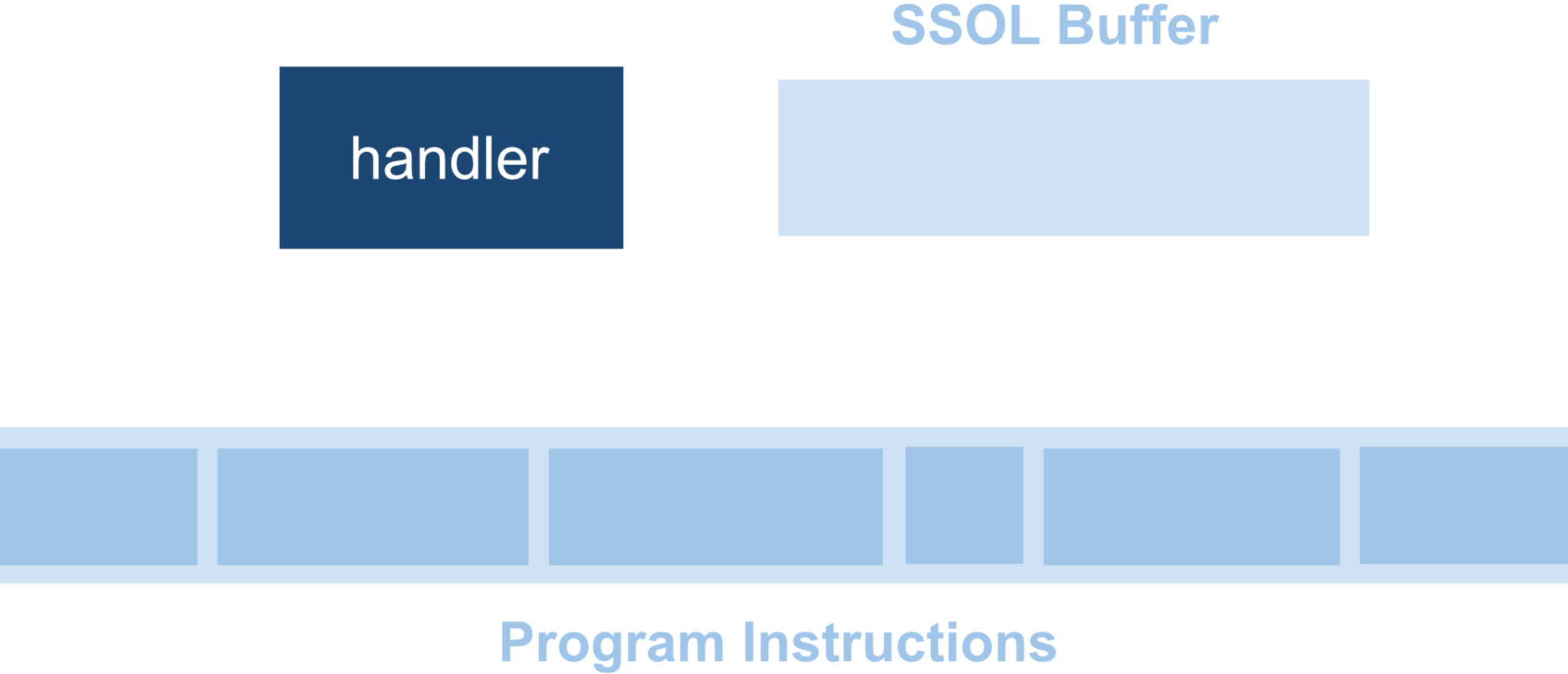
Kernel

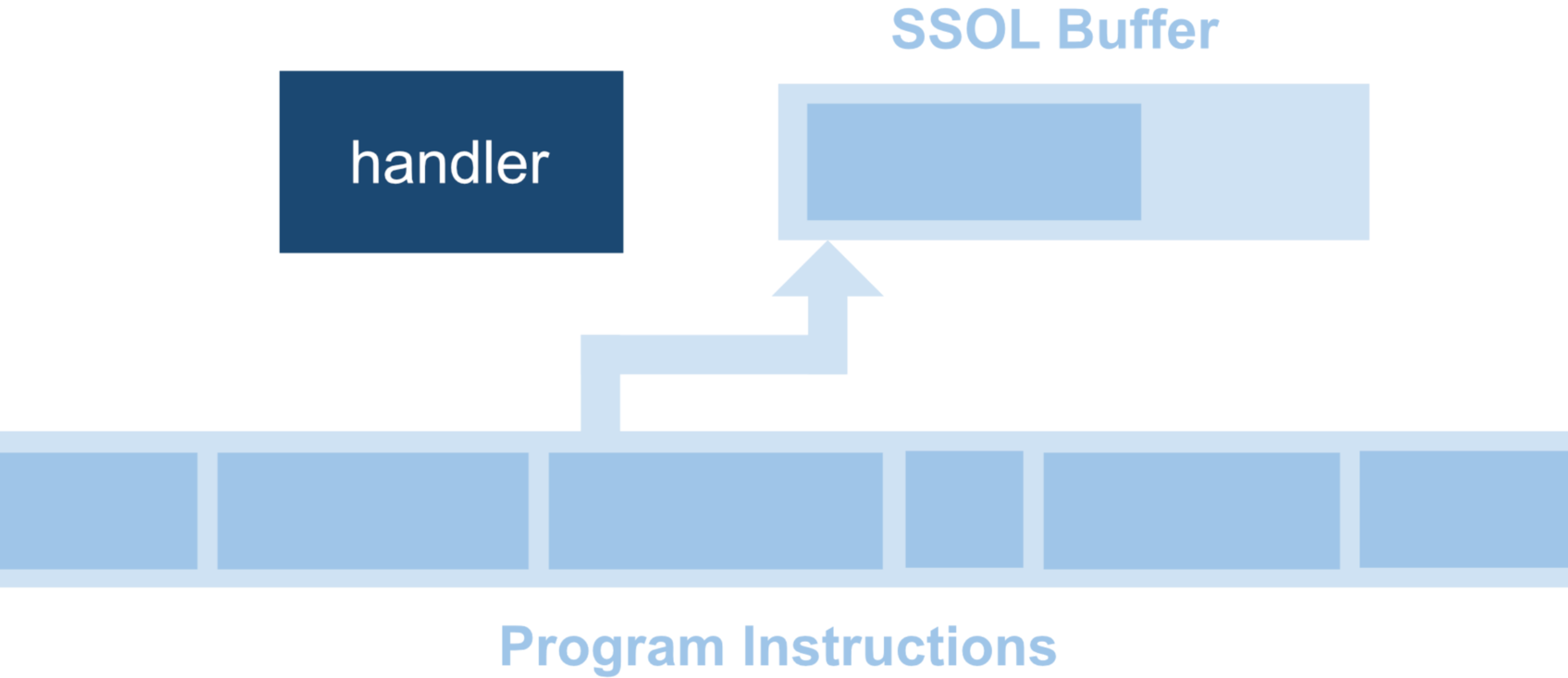
- kprobes - dynamic
- tracepoints - static

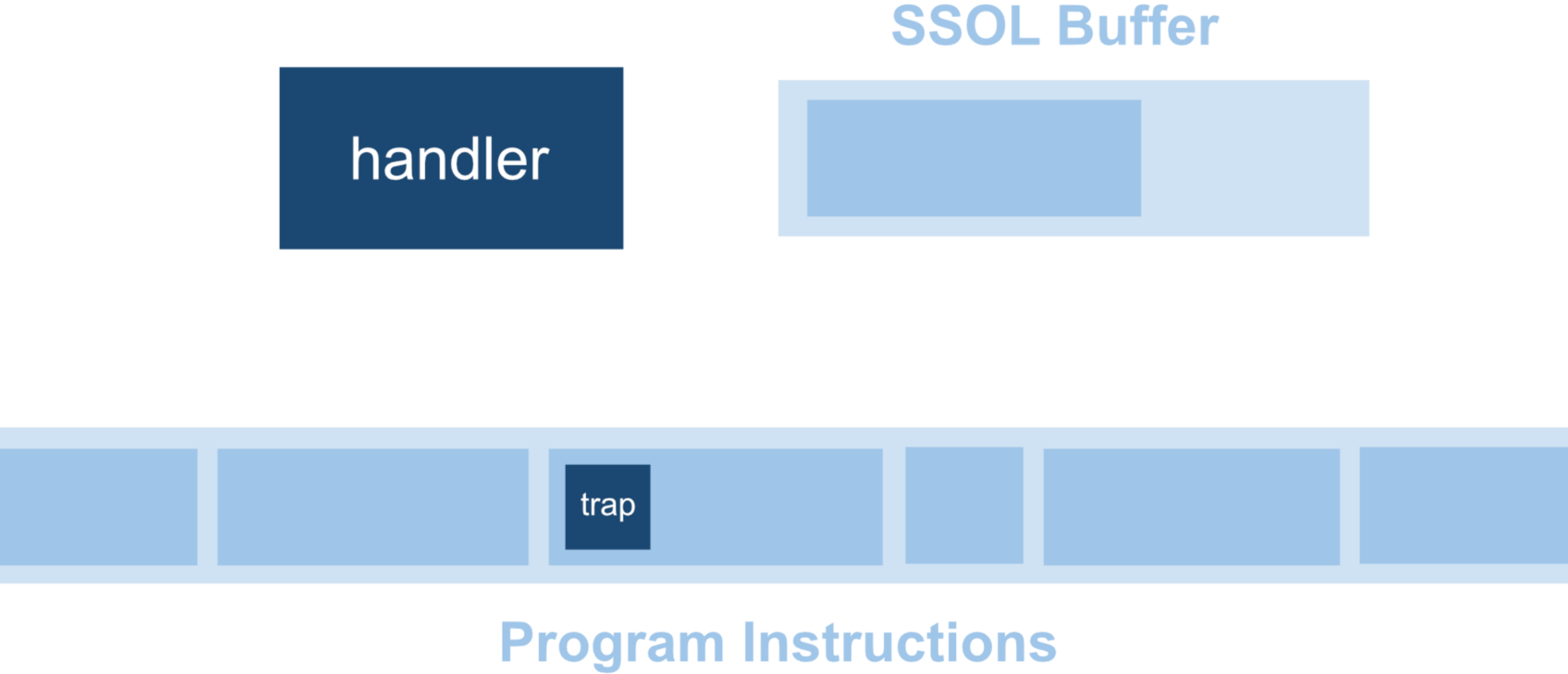
Other

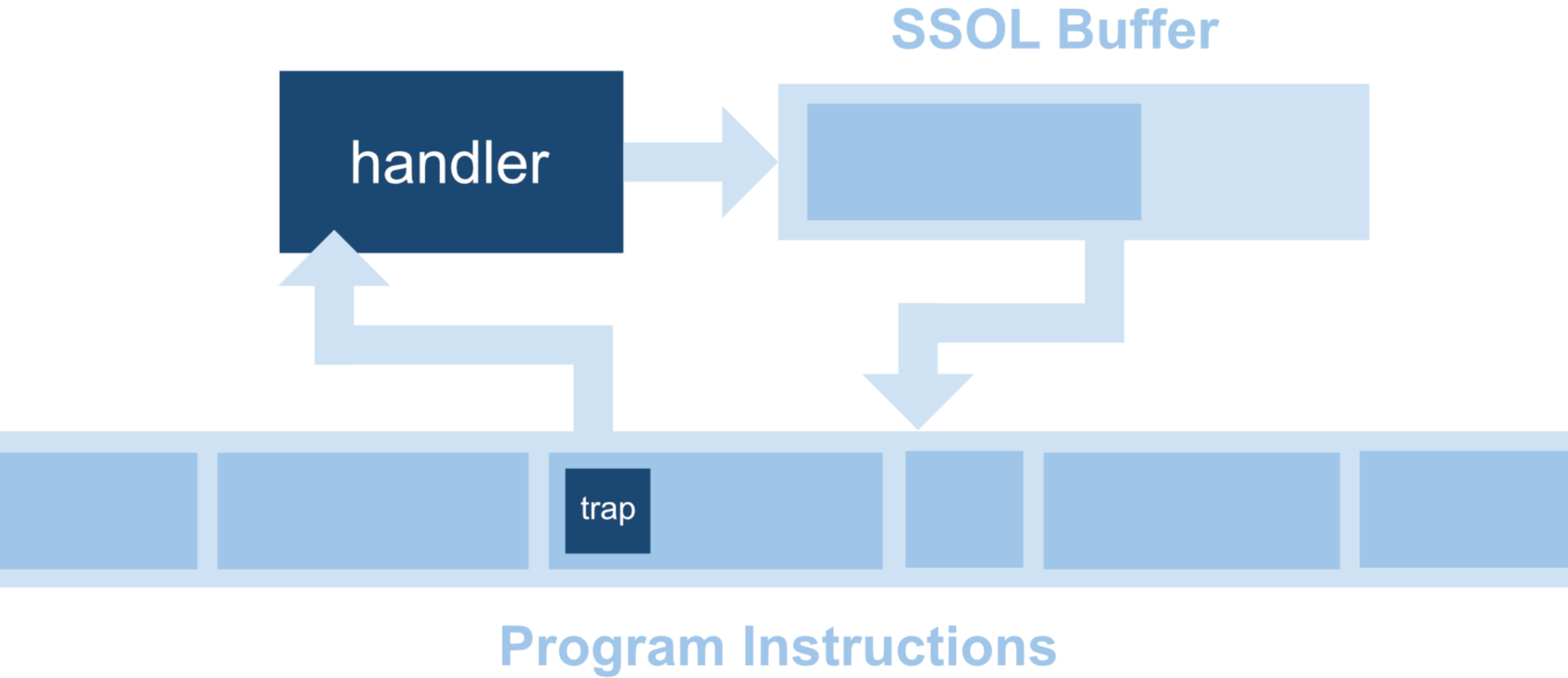
- sockets
- tc
- perf events
- etc

uprobes allows you to trace any instruction in user land with much **less overhead** than ptrace


SSOL Buffer
handler
Program Instructions


SSOL Buffer
handler
Program Instructions


SSOL Buffer
handler
trap
Program Instructions


SSOL Buffer
handler
trap
Program Instructions

USDT

- Tracepoints that are defined in advance by the developer
- They are typically used as tracing landmarks that are stable across time
- Can report arbitrary data when they fire
  - Kind of like logging but without always paying the performance cost
- Supported in most language runtimes (Java, Python, Node, Ruby)
  - This allows you to trace functions in dynamic languages by attaching to probes such as function__entry and function__return.
- Implemented in linux using uprobes

How do you write BPF programs?

BCC

github.com/iovisor/bcc

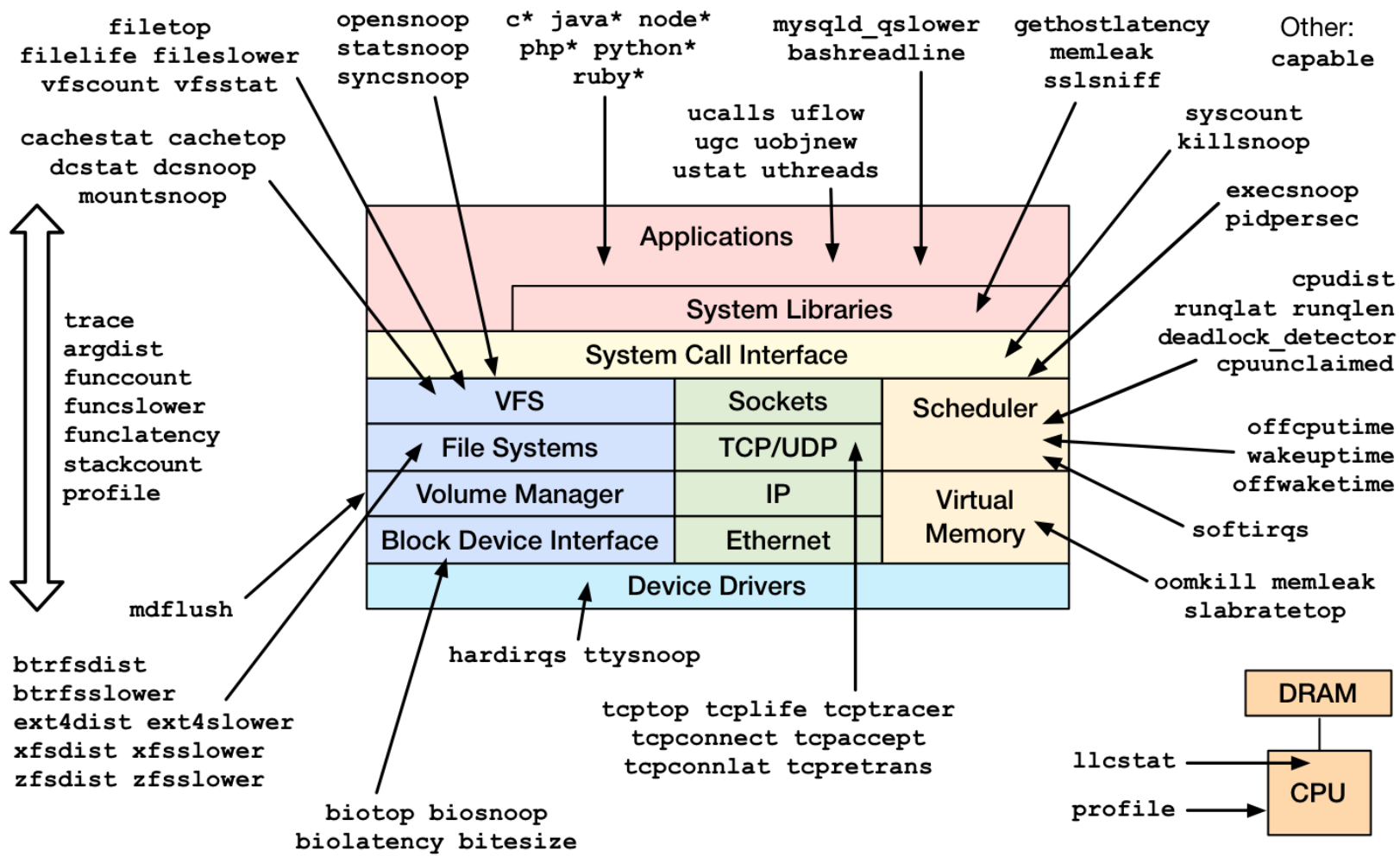
bpftrace

github.com/iovisor/bpftrace

BCC (BPF Compiler Collection)

- BCC is a compiler for BPF programs that are written in C
- It also assists with interacting with your BPF programs from user land
- It is implemented as a library (libbcc.so)
- This library has a lot of awesome functionality and is quite mature
- It comes with a collection of pre-built tools that are incredibly useful
- It also comes with bindings for Python and LUA
- Third party Go bindings exist

Linux bcc/BPF Tracing Tools



https://github.com/iovisor/bcc#tools 2017

uprobe demo

```
func f() {
        atomic.AddUint64(&count, 1)
}

func main() {
        for {
                f()
        }
}
```

```
bpf_text = r"""
BPF_ARRAY(count, u64, 1);

int do_trace() {
    count.increment(0);
    return 0;
};
"""

b = BPF(text=bpf_text)
b.attach_uprobe(name=sys.argv[1], sym="main.f", fn_name="do_trace")

count = b["count"]

while True:
    time.sleep(1)
    print("{:15,} ops/s".format(count[0].value))
    count.clear()
```

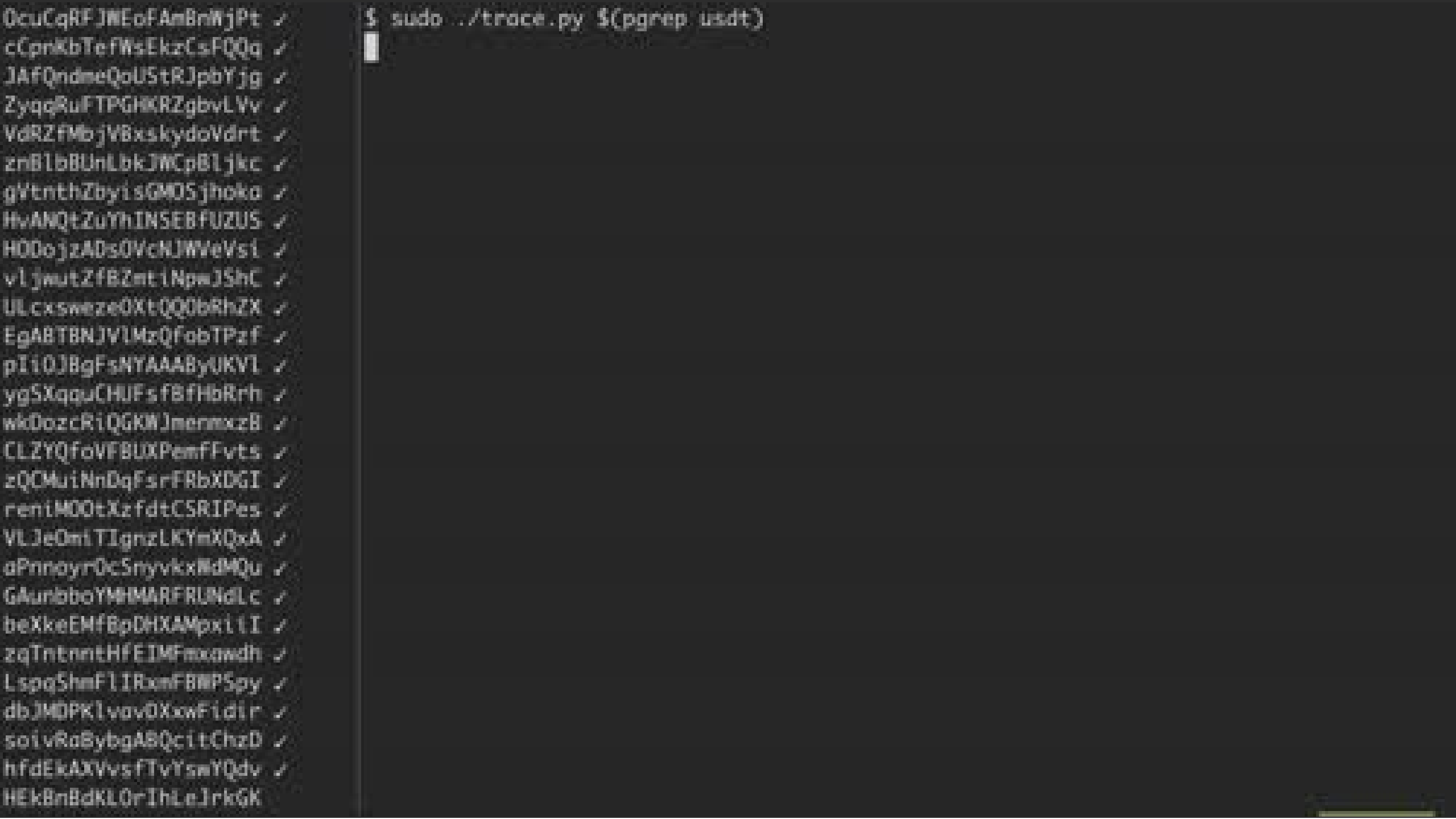

$ uprobes
$ sudo ./trace.py $(which uprobes)

usdt demo

```
var (
        probes = salp.NewProvider("usdt")
        entry  = salp.MustAddProbe(probes, "entry")
        exit   = salp.MustAddProbe(probes, "exit")
)

func f() {
        entry.Fire()
        defer exit.Fire()
        http.Get("https://www.google.com/search?q=" + randStr())
}

func main() {
        salp.MustLoadProvider(probes)
        defer salp.UnloadAndDispose(probes)

        for {
                f()
        }
}
```

```
BPF_ARRAY(start, u64, 1);
BPF_HISTOGRAM(latency, u64);

int trace_entry() {
    u64 ts = bpf_ktime_get_ns();
    int zero = 0;
    start.update(&zero, &ts);

    return 0;
};

int trace_exit() {
    u64 *tsp;
    int zero = 0;

    // fetch timestamp and calculate delta
    tsp = start.lookup(&zero);
    if (tsp == 0 || *tsp == 0) return 0; // missed start
    u64 delta = (bpf_ktime_get_ns() - *tsp) / 1000000;

    // store as histogram
    latency.increment(bpf_log2(delta));
    start.delete(&zero);

    return 0;
};
```

```
u = USDT(pid=int(sys.argv[1]))
u.enable_probe(probe="entry", fn_name="trace_entry")
u.enable_probe(probe="exit", fn_name="trace_exit")
b = BPF(text=bpf_text, usdt_contexts=[u])

try:
    time.sleep(99999999)
except KeyboardInterrupt:
    b["latency"].print_log2_hist("milliseconds")
```


OcuCqRFJWEoFAmBnWjPt ✓
cCprKbTefWsEkzCsFQQq ✓
JAfQndmeQoUStRJpbYjg ✓
ZyqqRuFTPGHKRZgbvLVv ✓
VdRZfMbjVBxskydoVdrt ✓
znBlbBUnLbkJWCpBljkc ✓
gVtnthZbyisGMOSjhoka ✓
HvANQtZuYhINSEBfUZUS ✓
HODojzADsOVcNJWVeVsi ✓
vljwutZfBZmtiNpwJShC ✓
ULcxswezeOXtQQObRhZX ✓
EgABTBNJVlMzQfobTPzf ✓
pIiOJBgFsNYAAAByUKVl ✓
ygSXqquCHUFsfBfHbRrh ✓
wkDozcRiQGKWJmermxzB ✓
CLZYQfoVFBUXPemfFvts ✓
zQCMuiNnDqFsrFRbXDGI ✓
rentMOOtXzfdtCSRIPes ✓
VLJeOmiTIgnzLKYmXQxA ✓
aPnnoyrOcSnyvkxWdMQu ✓
GAunbboYMHMARFRUNdLc ✓
beXkeEMfBpDHXAMpxiiI ✓
zqTntnntHfEIMFmxawdh ✓
LspqShmFlIRxmFBWPSpy ✓
dbJMDPKlvavOXxwFidir ✓
soivRaBybgABQcitChzD ✓
hfdEkAXVvsfTvYswYQdv ✓
HEkBnBdKLOrIhLeJrkGK
$ sudo ./trace.py $(pgrep usdt)

bpftrace

simplifies writing these programs

```
uprobe:/path/to/bin:"main.f" {
    @ = count();
}

interval:s:1 {
    print(@);
    clear(@);
}
```

```
bpf_text = r"""
BPF_ARRAY(count, u64, 1);

int do_trace() {
    count.increment(0);
    return 0;
};
"""

b = BPF(text=bpf_text)
b.attach_uprobe(name=sys.argv[1],
    sym="main.f", fn_name="do_trace")

count = b["count"]

while True:
    time.sleep(1)
    print("{:15,} ops/s".format(
        count[0].value))
    count.clear()
```

```
usdt:/path/to/bin:entry {
    @start = nsecs;
}

usdt:/path/to/bin:exit {
    @ = hist(nsecs - @start);
    delete(@start);
}
```

```
bpf_text = r"""
BPF_ARRAY(start, u64, 1);
BPF_HISTOGRAM(latency, u64);

int trace_entry() {
    u64 ts = bpf_ktime_get_ns();
    int zero = 0;
    start.update(&zero, &ts);

    return 0;
};

int trace_exit() {
    u64 *tsp;
    int zero = 0;

    // fetch timestamp and calculate delta
    tsp = start.lookup(&zero);
    if (tsp == 0 || *tsp == 0) return 0; // missed start
    u64 delta = (bpf_ktime_get_ns() - *tsp) / 1000000;

    // store as histogram
    latency.increment(bpf_log2(delta));
    start.delete(&zero);

    return 0;
};
"""

u = USDT(pid=int(sys.argv[1]))
u.enable_probe(probe="entry", fn_name="trace_entry")
u.enable_probe(probe="exit", fn_name="trace_exit")
b = BPF(text=bpf_text, usdt_contexts=[u])

try:
    time.sleep(99999999)
except KeyboardInterrupt:
    b["latency"].print_log2_hist("milliseconds")
```

sysdig
system tap
lttng
dtrace for linux
ktap
ply

USDT
uprobes
perf events
tracepoints
PMCs
kprobes
perf

trace compass
catapult
trace-cmd
kernel shark
ftrace
ptrace

BCC

github.com/iovisor/bcc

bpftrace

github.com/iovisor/bpftrace

docker

cgroups
namespaces
seccomp

bpftrace

ebpf
uprobes
kprobes
tracepoints
perf_events

- ✓ Intercepting at any point of execution
- ✓ Without restarting the process
- ✓ With as low overhead as possible
- ✓ Read from memory and registers
- ✓ Collect data across multiple processes and the kernel
- ✓ And do it all safely

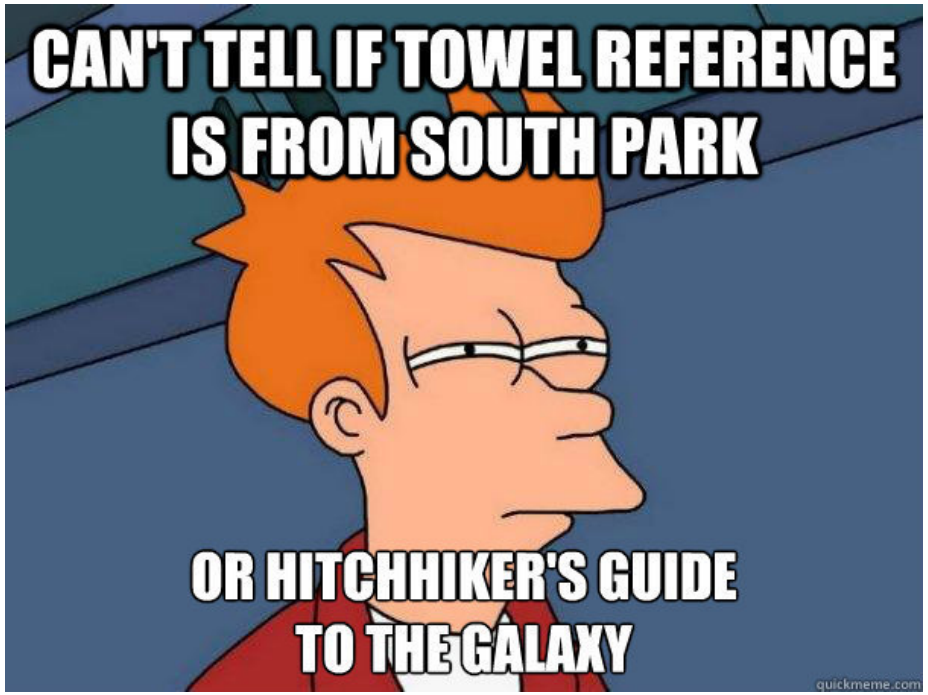
Method

Tools

Practice

We need to Deploy a Container to Probe our Applications

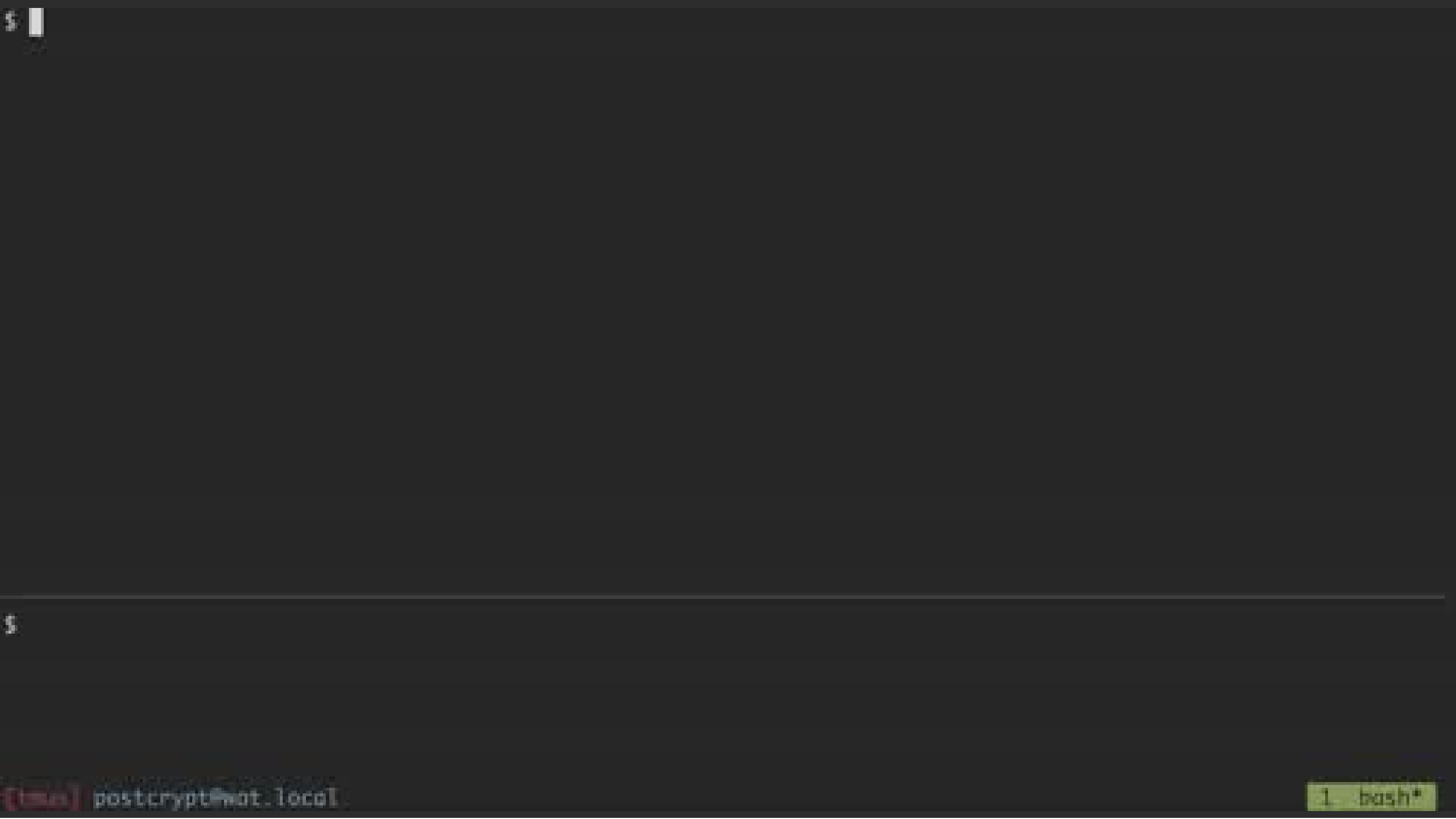
github.com/jasonkeene/towel



docker image
daemonset
kubectl plugin

```
spec:
  # share host pid namespace
  hostPID: true
  containers:
  - name: towel
    image: jasonkeene/towel
    securityContext:
      # run as root
      privileged: true
    volumeMounts:
    - name: sys
      mountPath: /sys
    - name: libmodules
      mountPath: /lib/modules
    - name: varlibdocker
      mountPath: /var/lib/docker
    - name: varrun
      mountPath: /var/run
```

```
  volumes:
  # kernel/debug/tracing
  - name: sys
    hostPath:
      path: /sys

  # kernel headers
  - name: libmodules
    hostPath:
      path: /lib/modules

  # container file systems
  - name: varlibdocker
    hostPath:
      path: /var/lib/docker

  # docker.sock
  - name: varrun
    hostPath:
      path: /var/run
```


$
$
[tmux] postcrypt@mat.local
1 bash*

This runs as **root!**

Make sure you delete the daemonset when it is no longer needed.

Also, put the daemonset in a namespace that is restricted.

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: secret-namespace
  name: exec-towel
rules:
# ...
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create"]
```

```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: secret-namespace
  name: jane-exec-towel
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: exec-towel
  apiGroup: rbac.authorization.k8s.io
```

How to Get Started?

tutorial at:

github.com/jasonkeene/towel

With these tools we can

Ask **Questions**, Get **Answers**

and

Better **Understand** our Systems

Thank You!

Jason Keene

Pivotal Software

k8s slack: @jasonkeene

github.com/jasonkeene