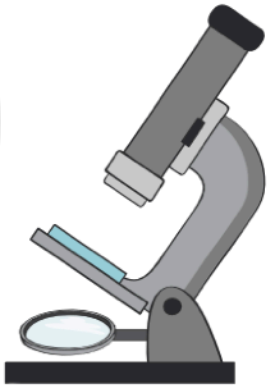
Dynamic Instrumentation for Go



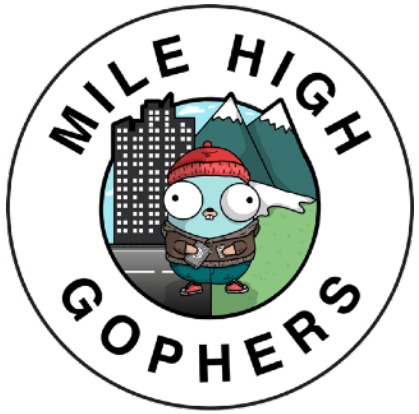
Hello!

I’m Jason Keene

Colorado Gophers

Pivotal

Coinbase



Pivotal

coinbase

Hello!

I'm Jason Keene

Colorado Gophers

Pivotal

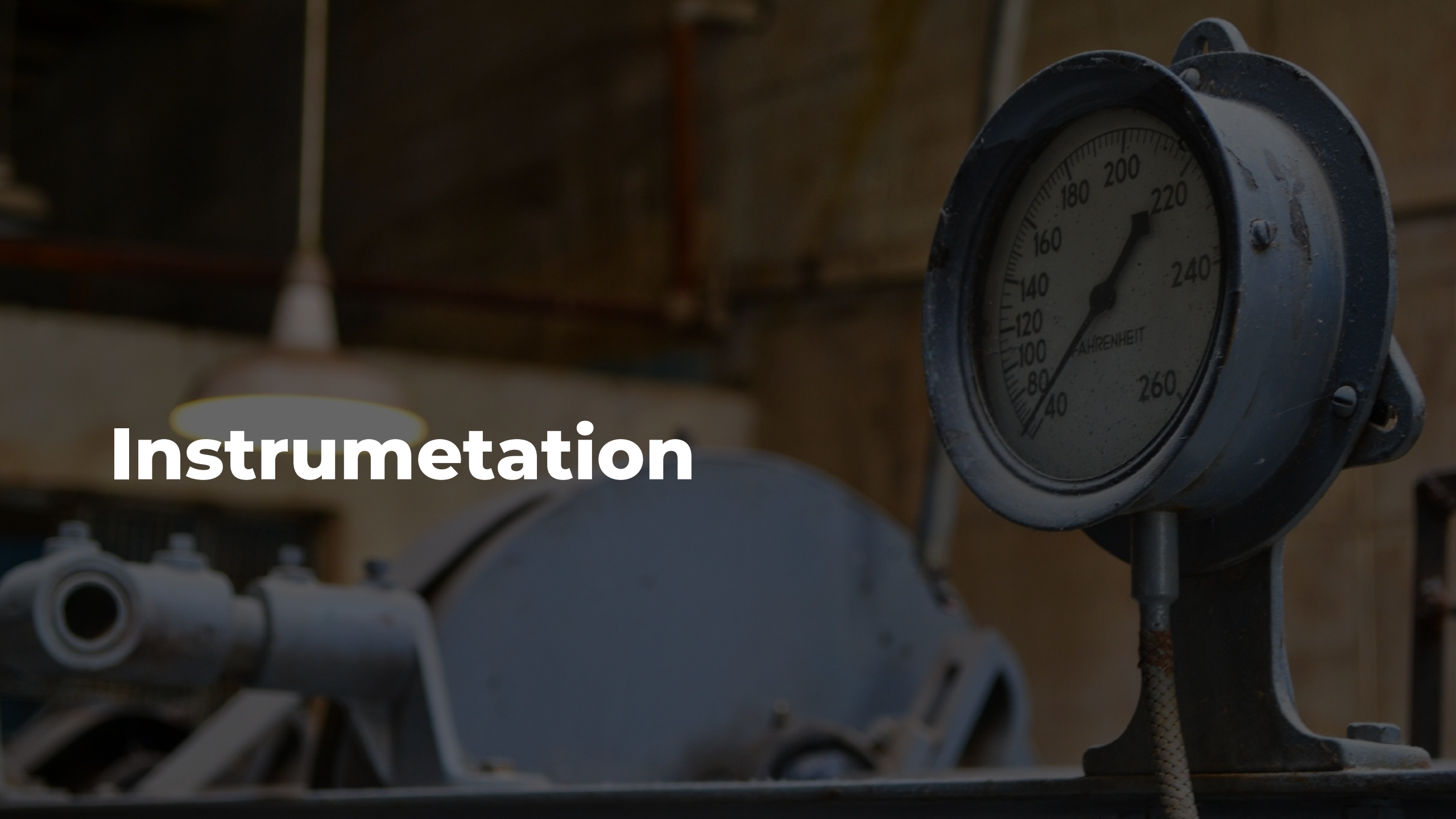
Coinbase

BPF

uprobes

ptrace

delve



Instrumetation

Instrumentation is anything that allows you to make measurements


0
20
40
60
80
100
120
140
160
180
km/h


DO NOT PAINT UNIT
NE PAS PEINDRE L'APPAREIL
ADDITIONAL MARKINGS ON BACK.
INSTRUCTIONS. ADDITIONAL
DE L'UTILISATEUR. INDICATIONS SUPPLÉMENTAIRES AU VERSO


It’s
Story
Time

Antonie van Leeuwenhoek

Dutch cloth merchant in the 1600s

He wanted a way to inspect the fibers of clothing he sold

He improved the design of the early microscope using lenses that allowed him to see clear, focused images of high magnification

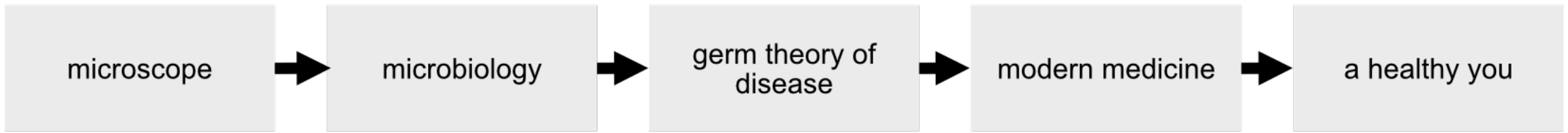
He used his microscope to study the water in Holland.

He discovered that the water contained little creatures, single-celled organisms that he called

Animalcules (Bacteria)

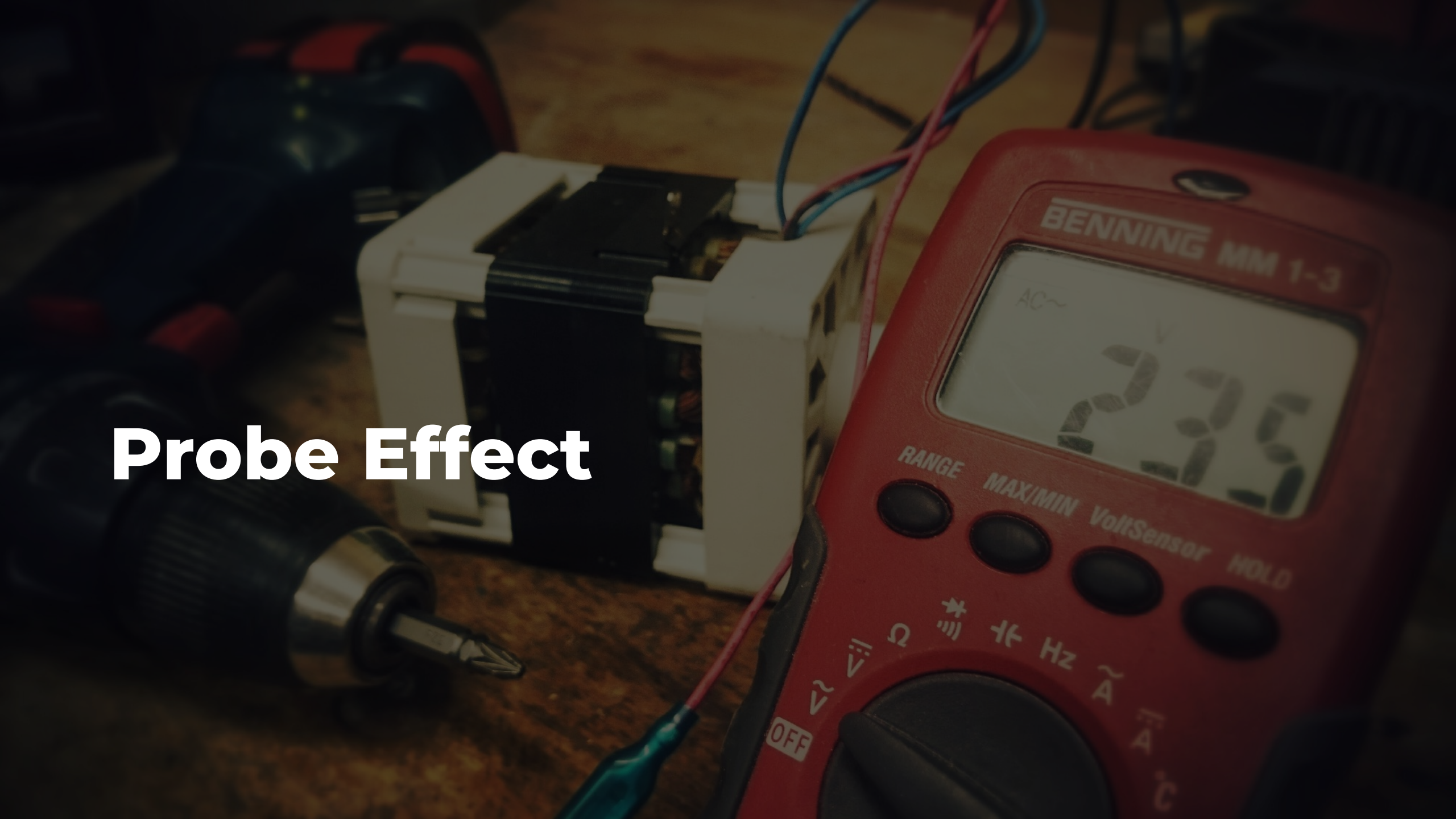
So why am I telling you about this?

The development of the microscope was required to make progress in biology.


microscope
microbiology
germ theory of disease
modern medicine
a healthy you

A new kind of instrumentation was required to achieve a new level of understanding in biology.

Instrumentation in Software



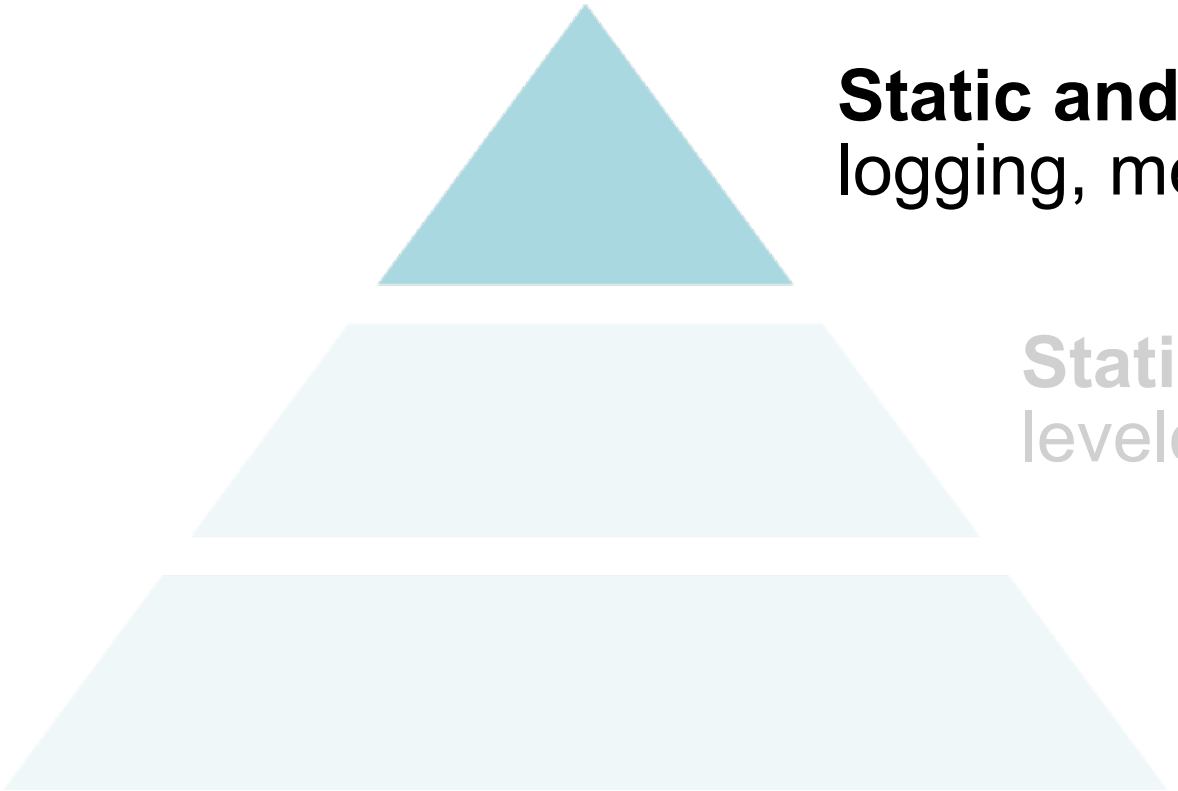
Probe Effect

Static and Always On
logging, metrics, distributed tracing

Static and Requires Activation
leveled logging, pprof, usdt probes

Dynamic Instrumentation
debuggers, tracers, uprobes/bpf

Measurement Coverage

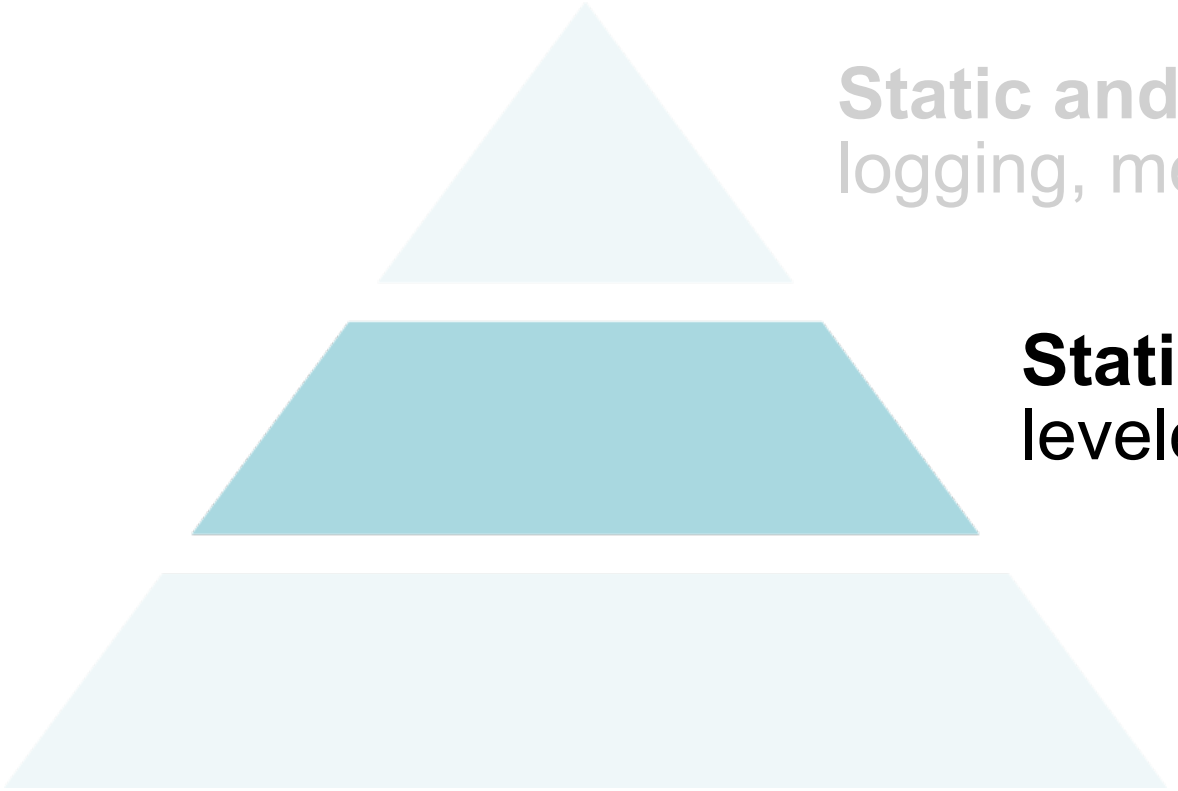


Static and Always On
logging, metrics, distributed tracing

Static and Requires Activation
leveled logging, pprof, usdt probes

Dynamic Instrumentation
debuggers, tracers, uprobes/bpf

Measurement Coverage



Static and Always On
logging, metrics, distributed tracing

Static and Requires Activation
leveled logging, pprof, usdt probes

Dynamic Instrumentation
debuggers, tracers, uprobes/bpf

Measurement Coverage

Static and Always On
logging, metrics, distributed tracing

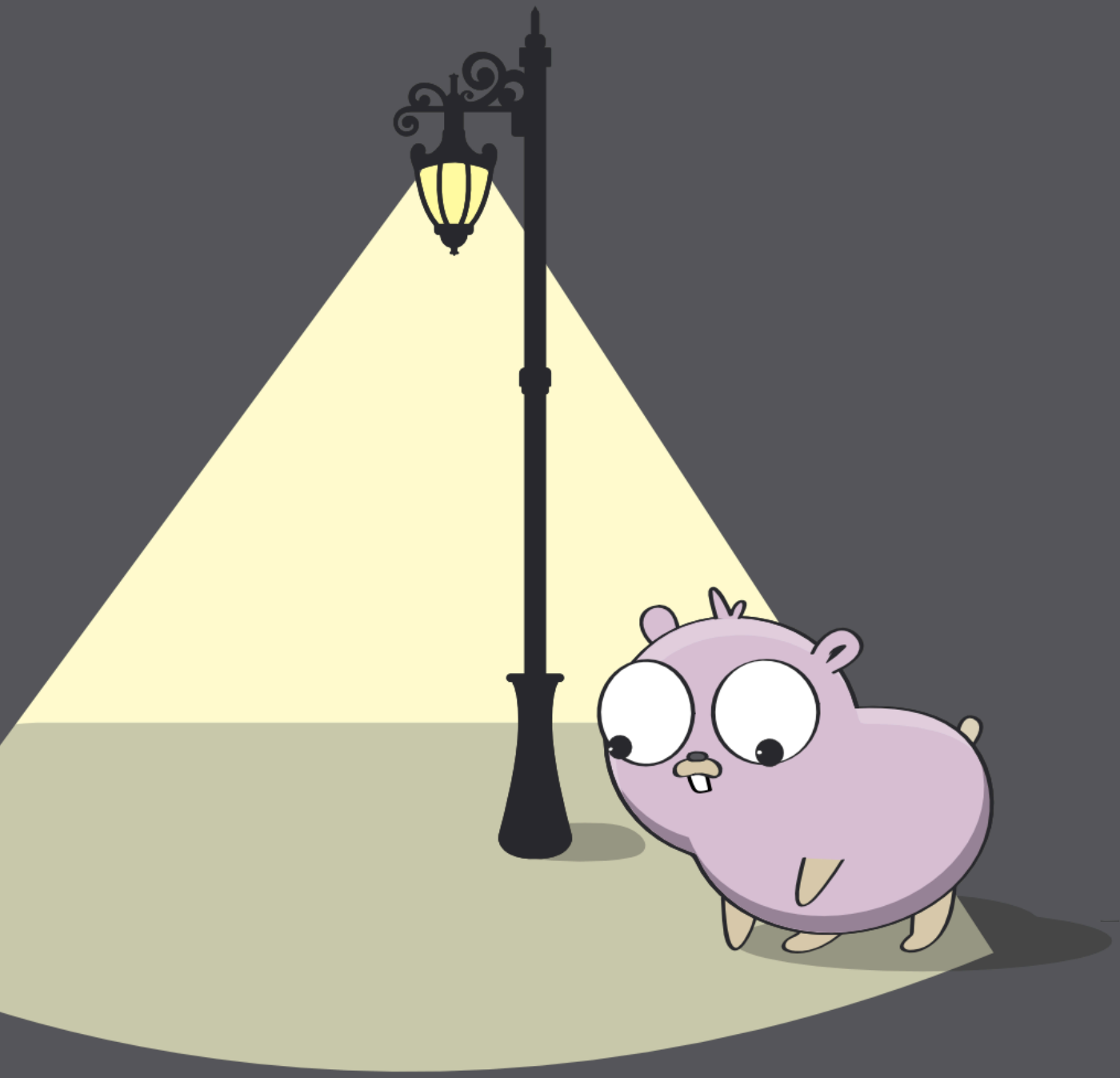
Static and Requires Activation
leveled logging, pprof, usdt probes

Dynamic Instrumentation
debuggers, tracers, uprobes/bpf

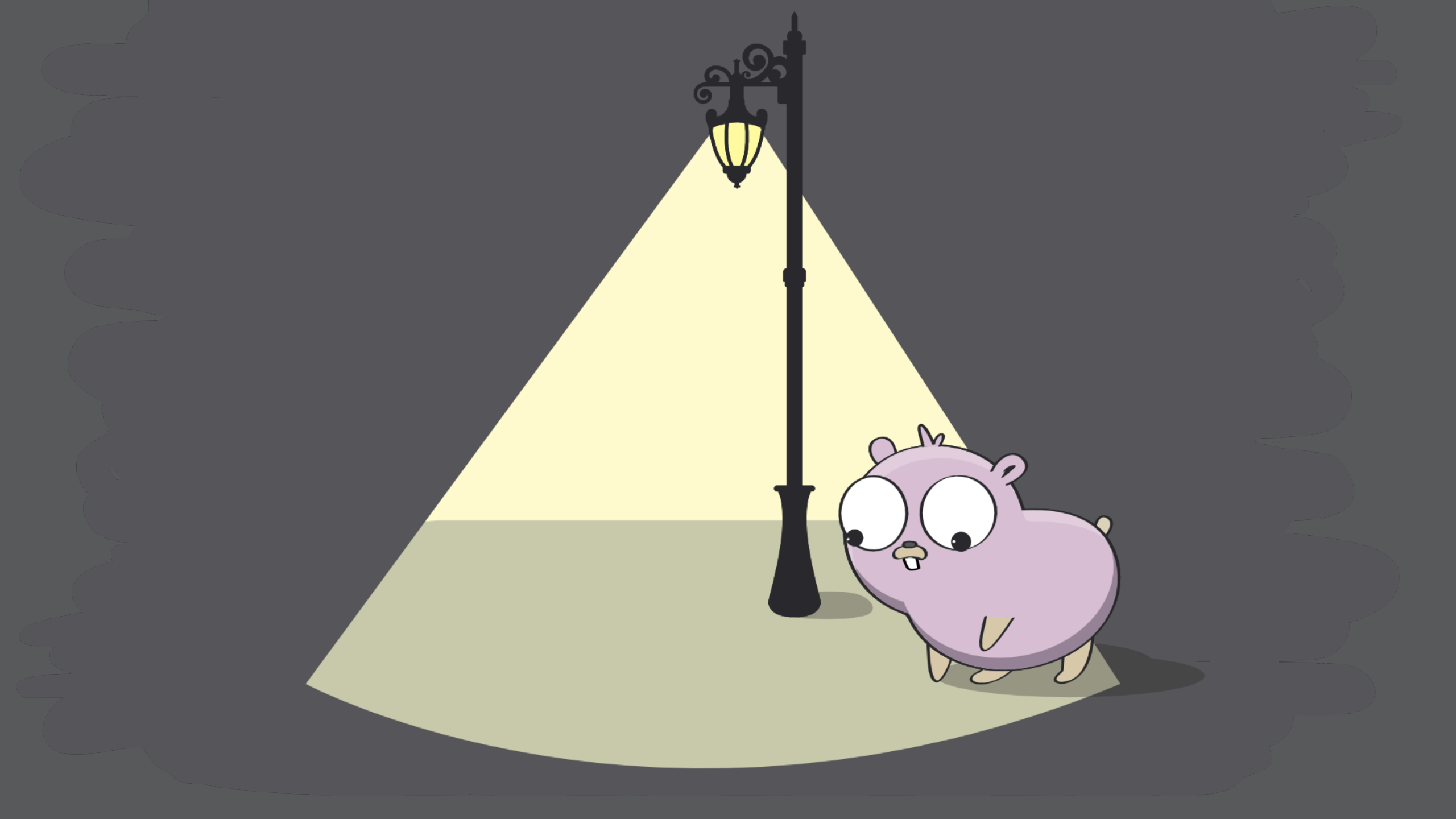
Measurement Coverage

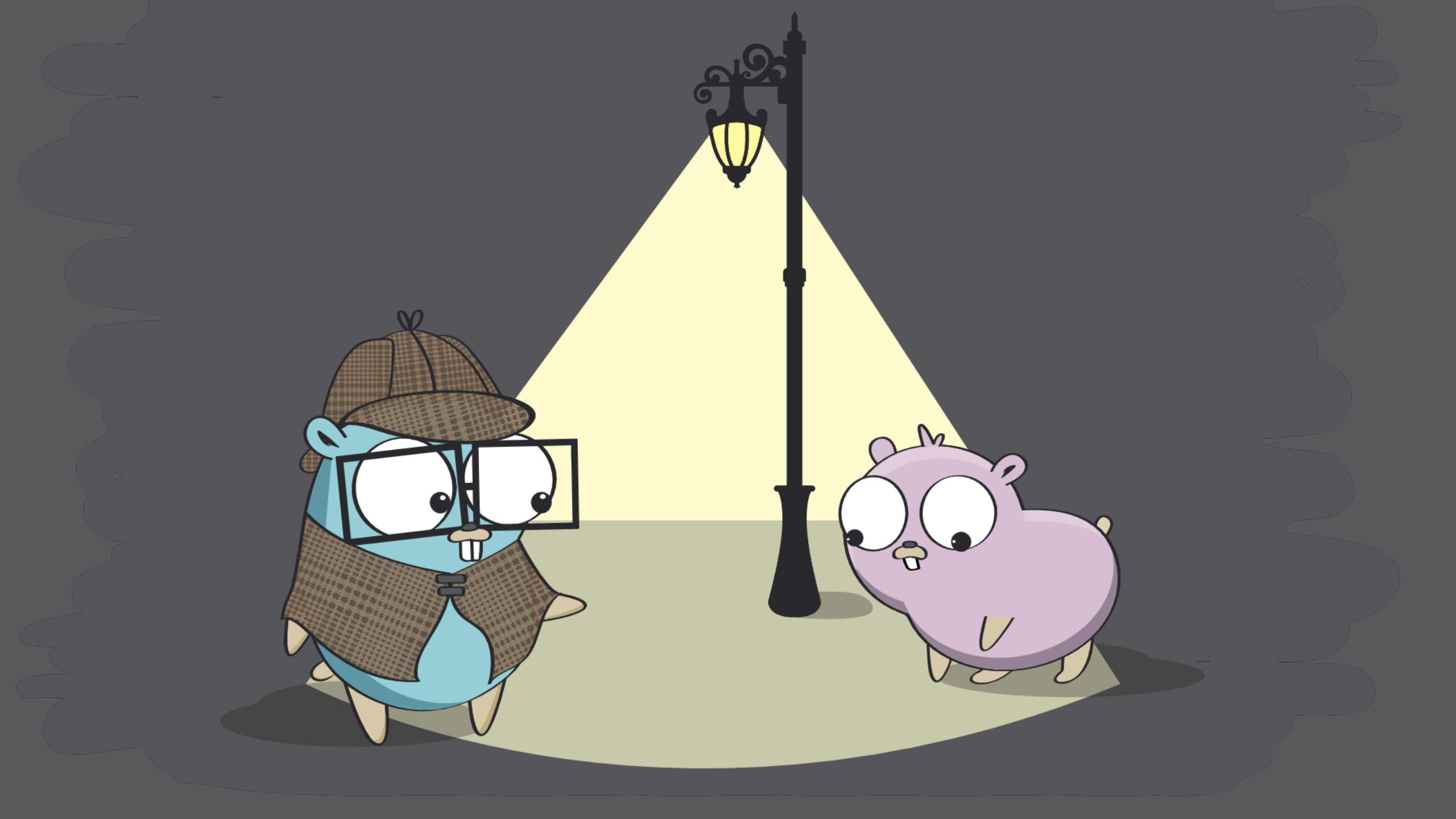
We want tools that can answer arbitrary questions about our software

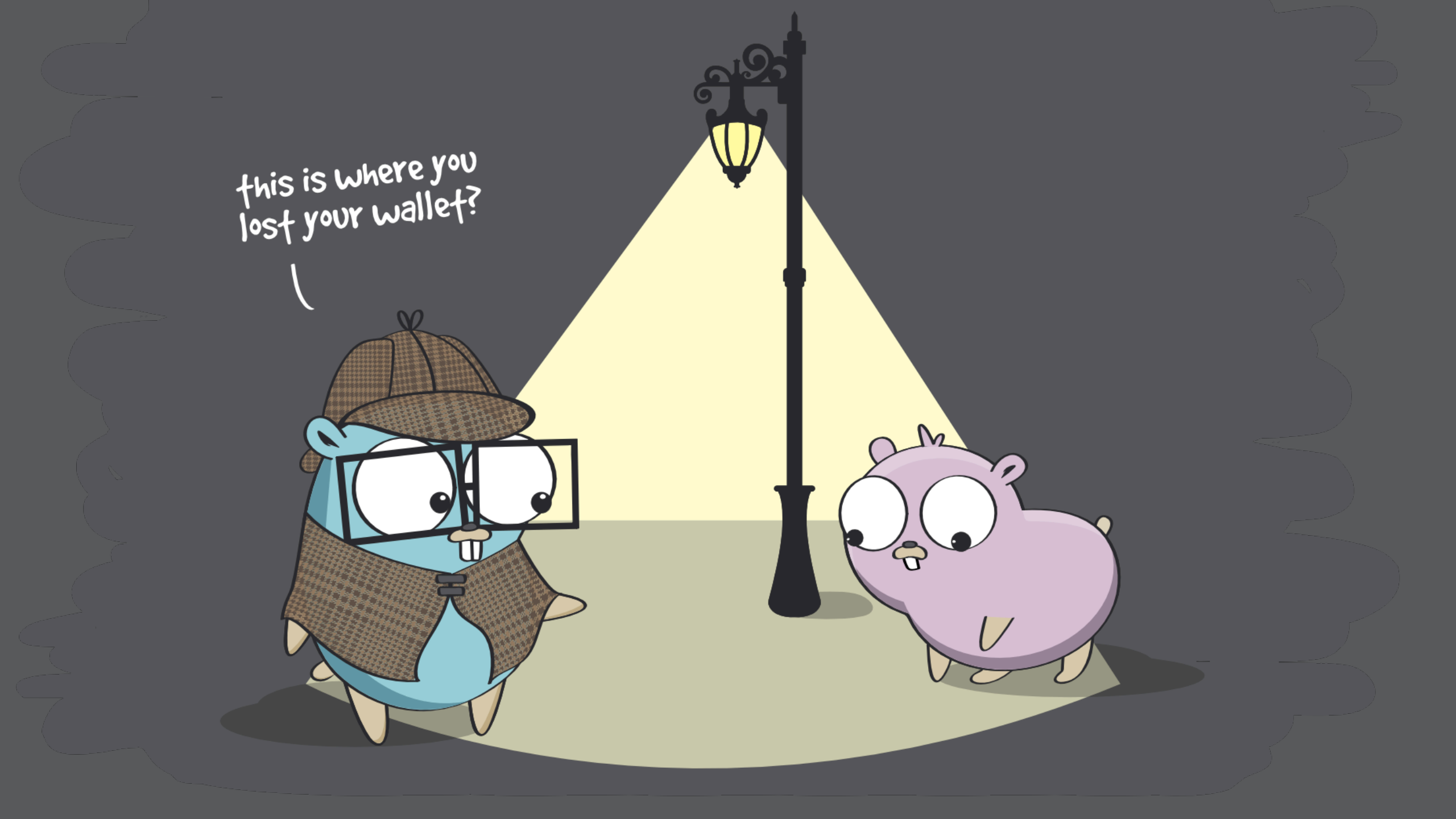
Intercept any point of execution
Without restarting the process
With low overhead
And do it all safely

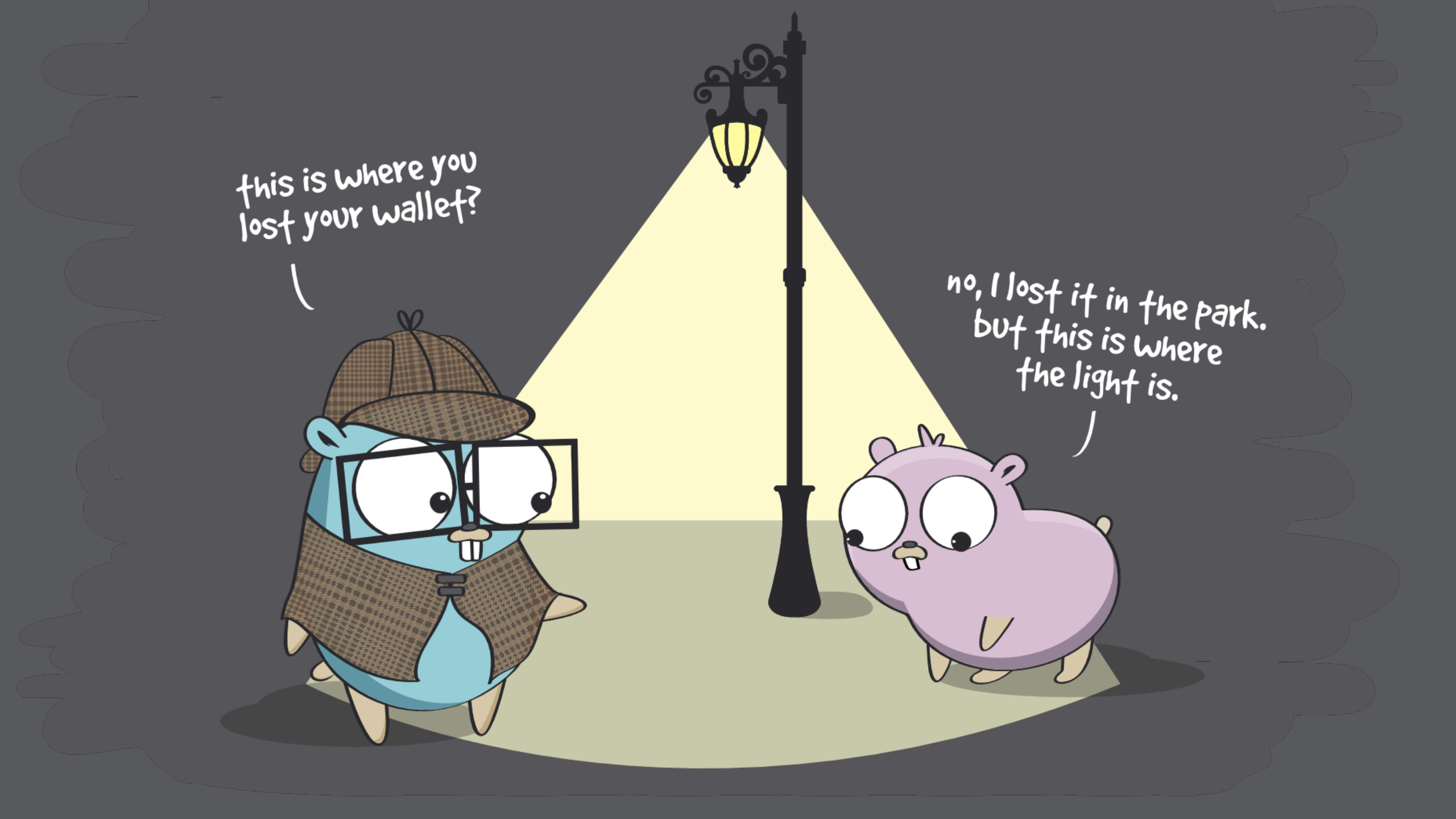


Streetlight Effect






this is where you lost your wallet?

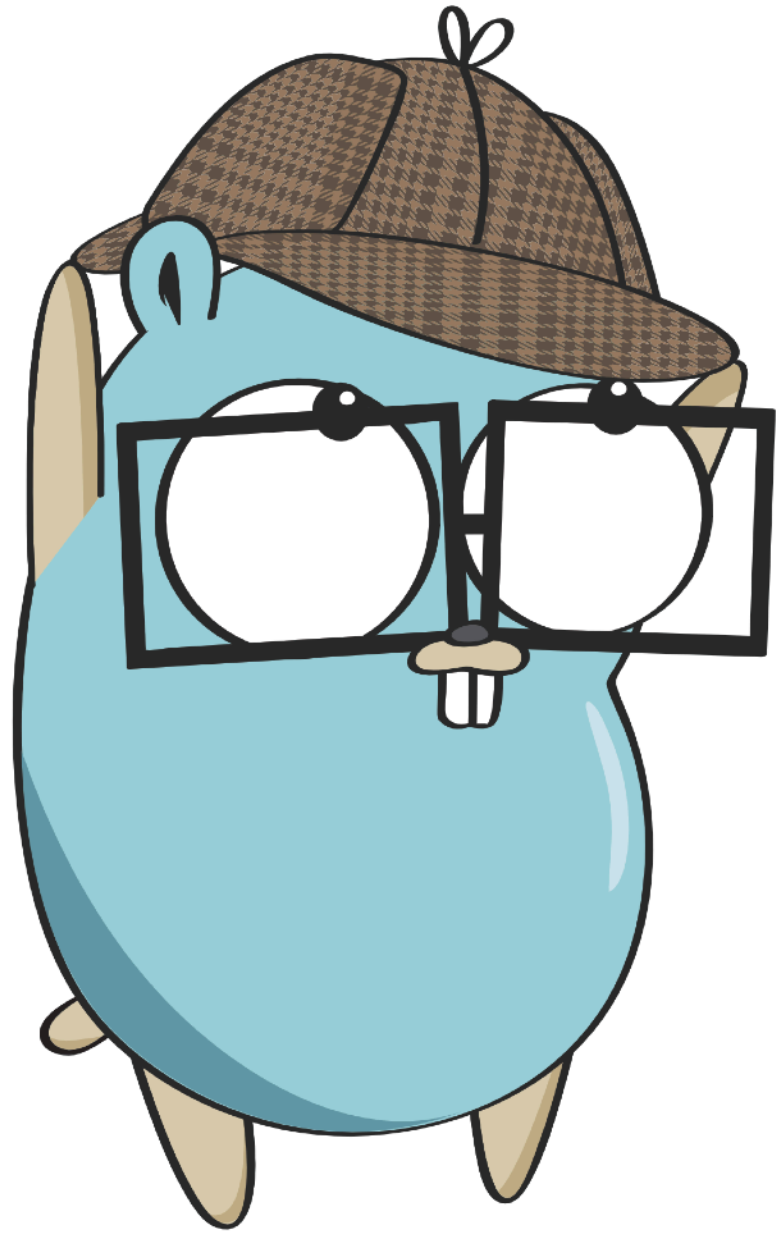

this is where you lost your wallet?
no, I lost it in the park. but this is where the light is.

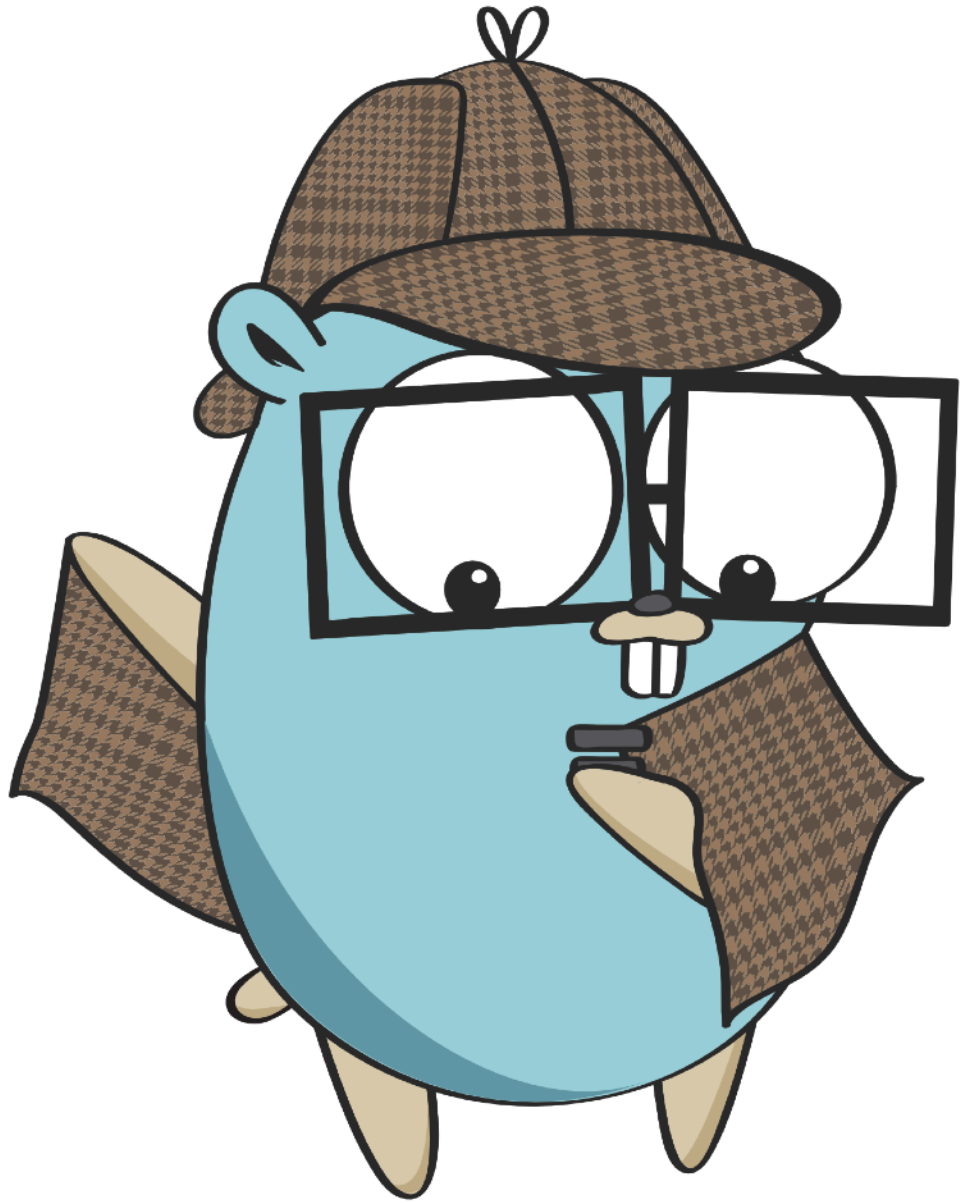
Dynamic Instrumentation
allows you to look in the park.

> *I want a toolkit that can examine running programs and answer questions about them. Questions I don't know until the problem arises...*
>
> *I'd like not to require compilation-time changes. I'd like to come to an arbitrary program while it's running...*
>
> *I want a programming interface above which I can build whatever is needed, either generally-useful tools or ad hoc probes. And the programming language I want to use for this is Go.*
>
> — *Rob Pike*

What Rob wants does not exist, yet...

However, there have been many advances that provide a good deal of capability.





ptrace and Debuggers

Debuggers are Incredibly Powerful Tools

Starting GDB

Debuggers Use the ptrace System Call

- Suspend/Continue Execution
- Single-step
- Read and Write to Memory
- Read and Write Registers
- Intercept Signals

Let’s see how debuggers use ptrace to set breakpoints and intercept execution

USER

KERNEL

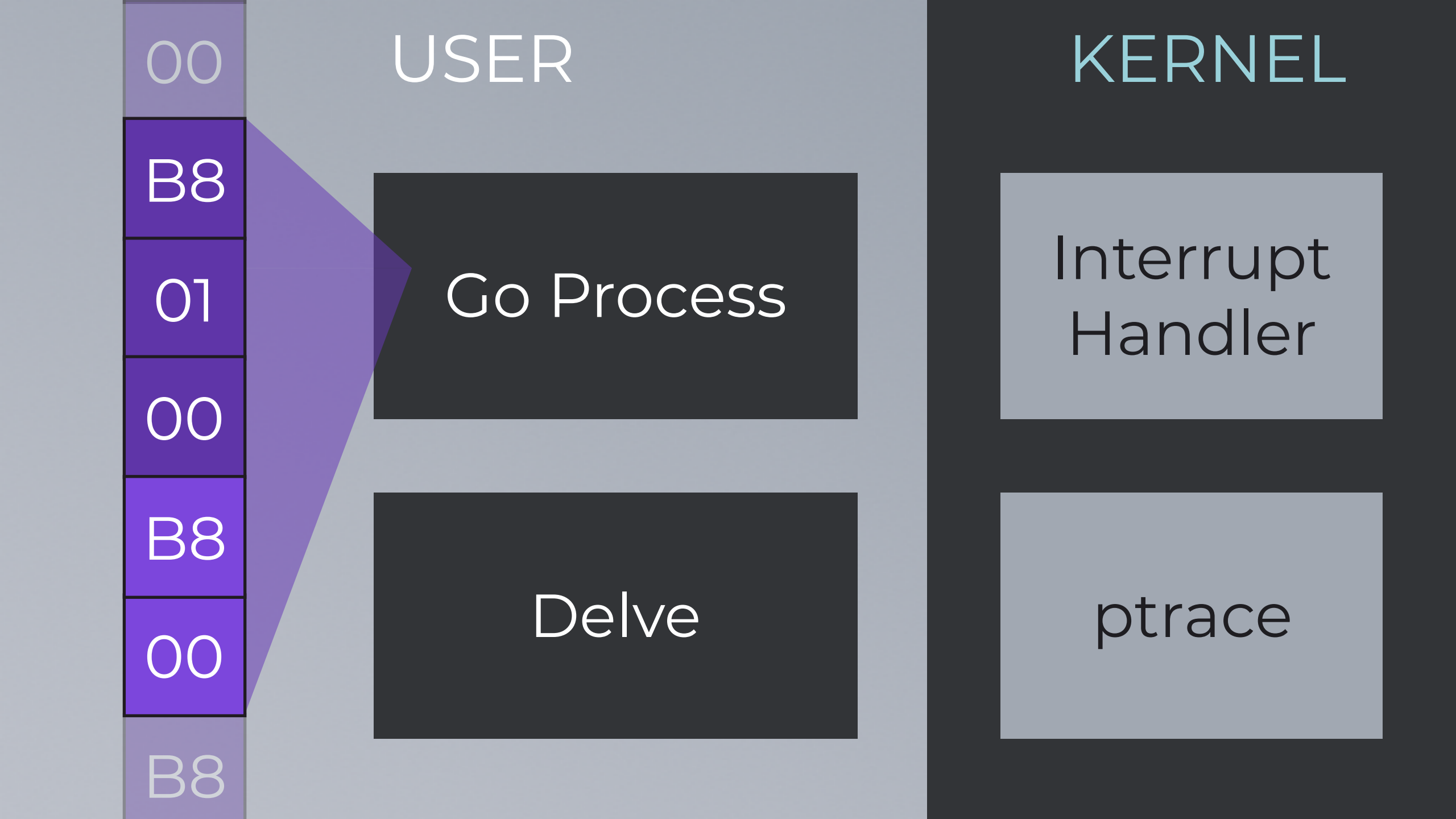
Go Process

Interrupt Handler

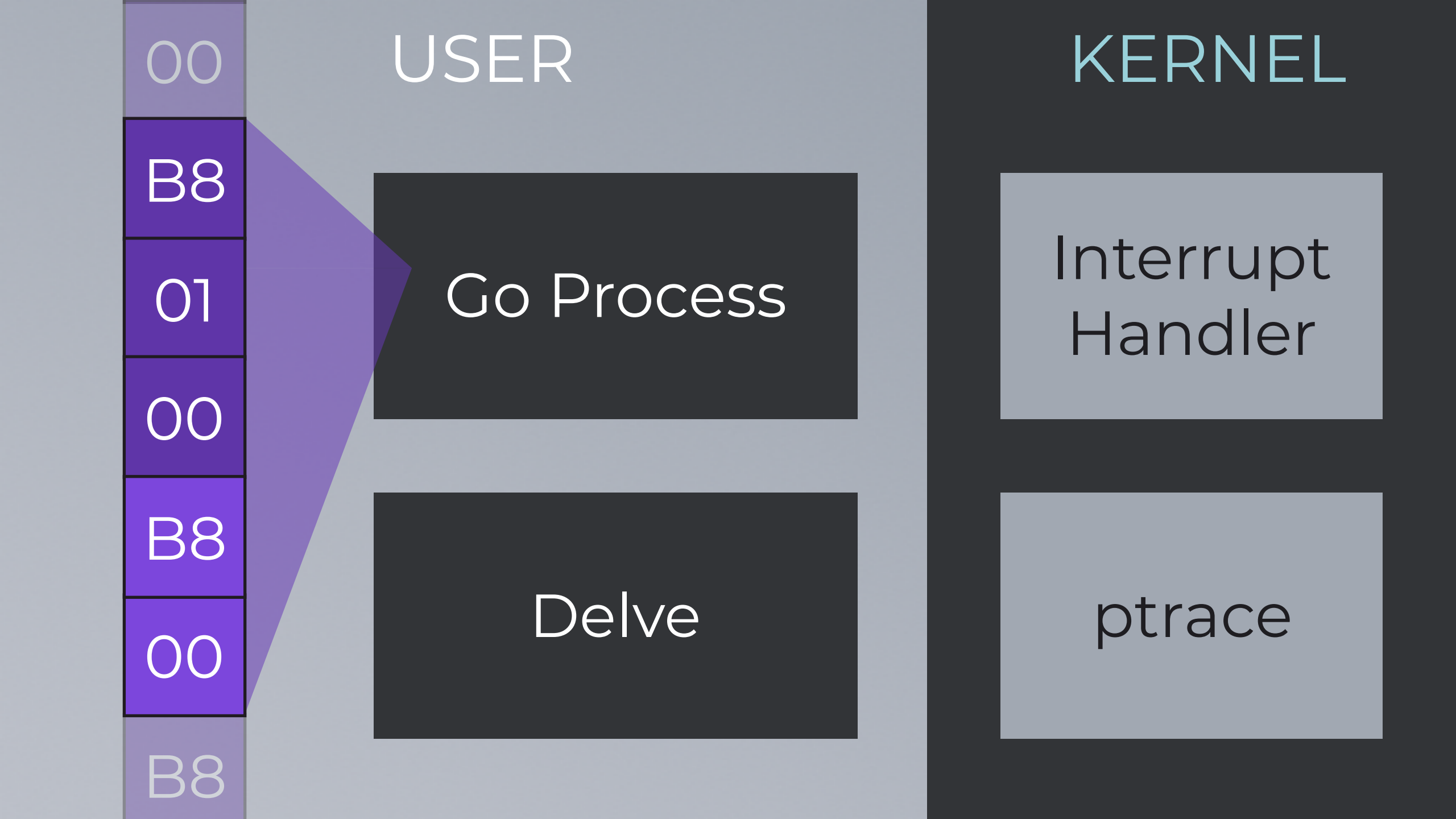
Delve

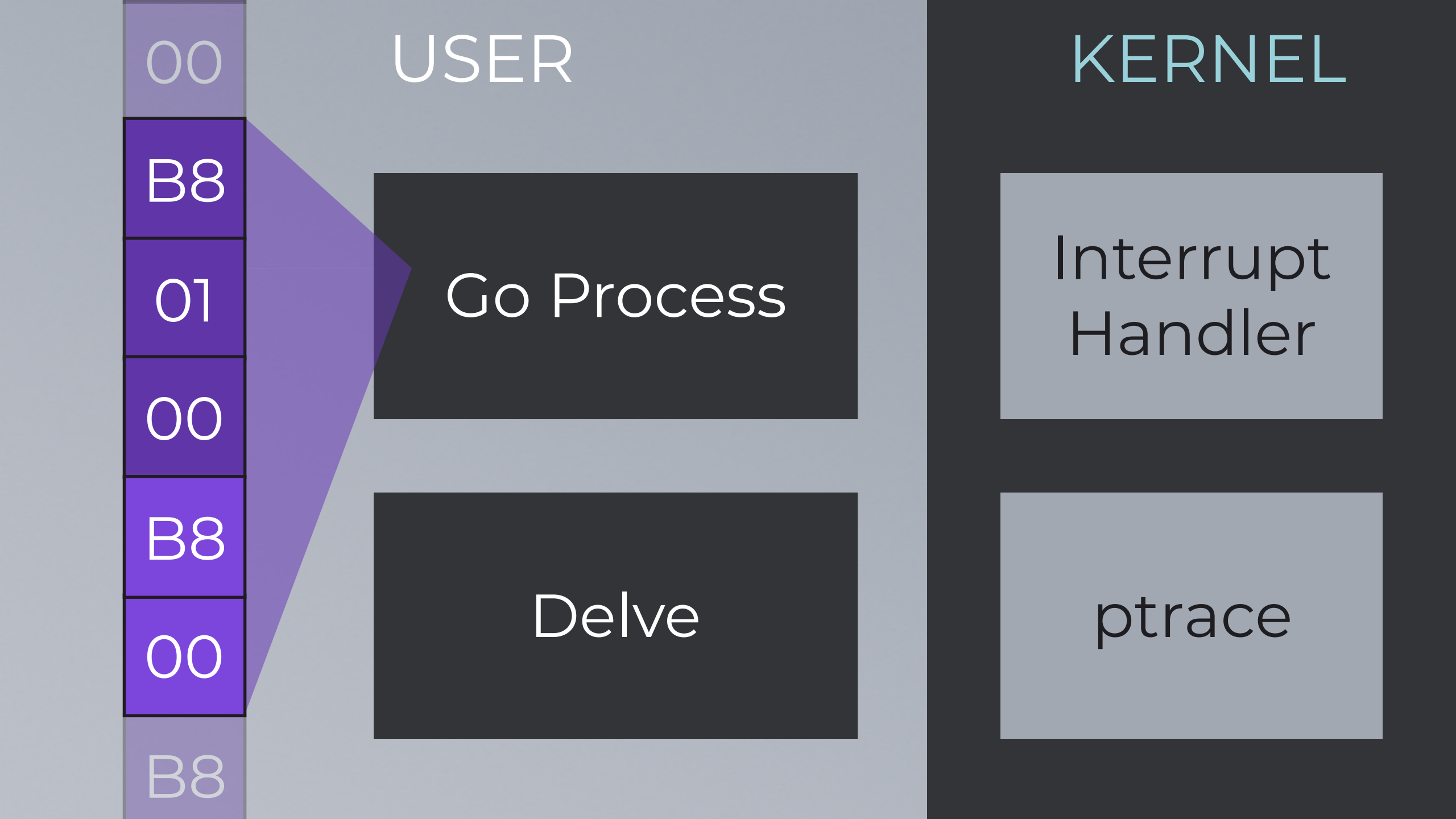
ptrace

Let’s Take a Look at the Instructions as the Program Runs

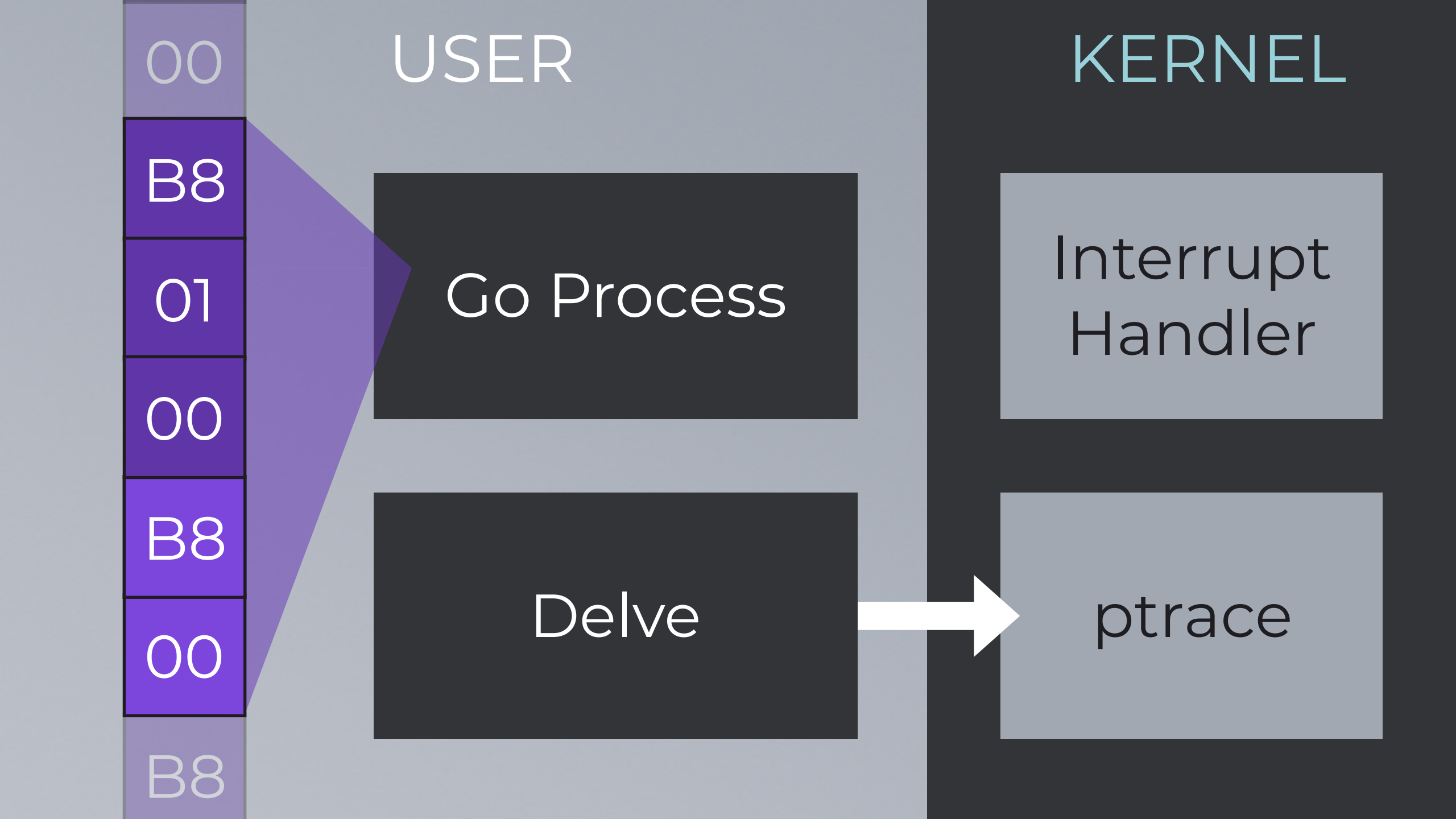

00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
Delve
Interrupt Handler
ptrace

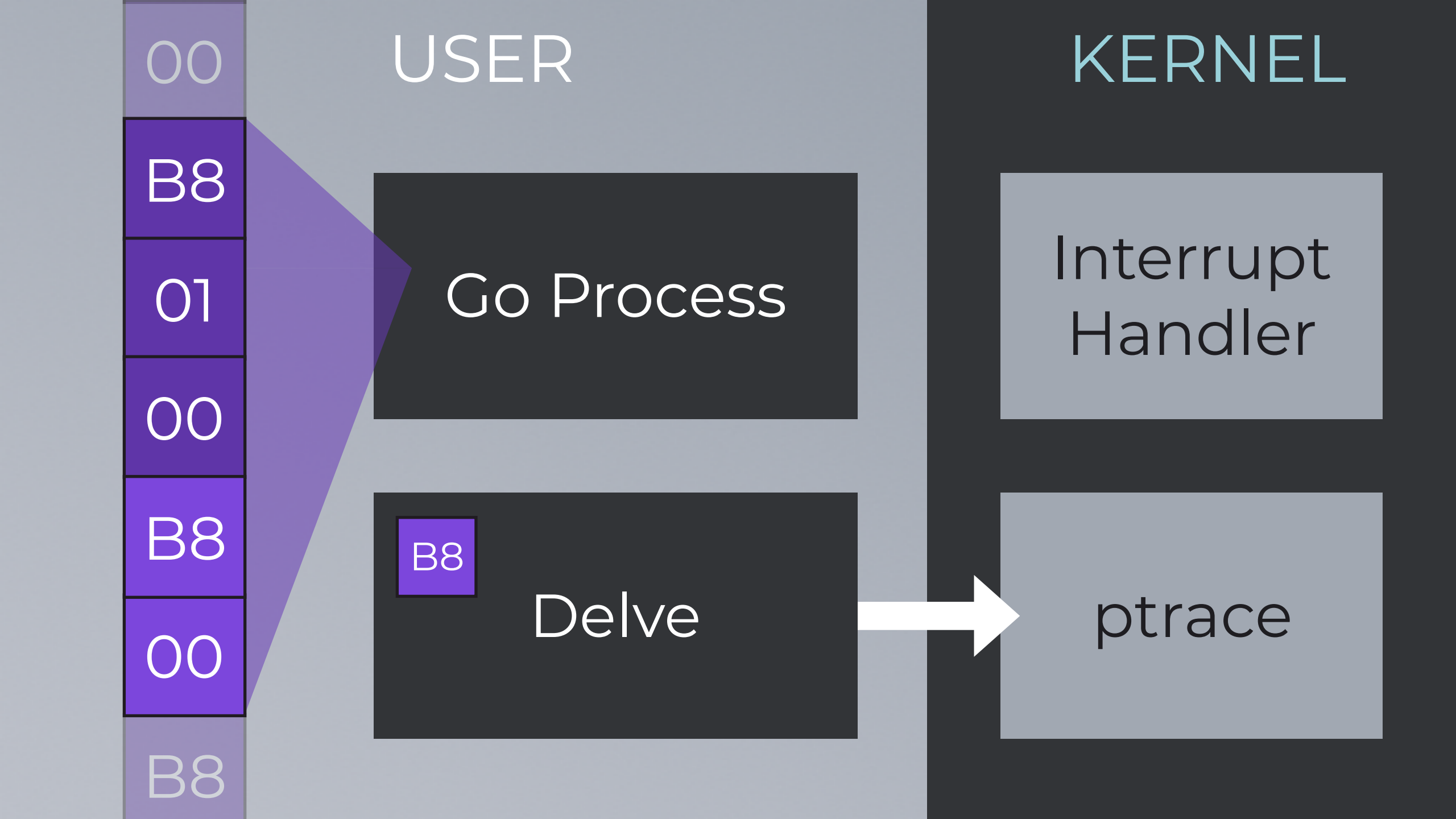

B8
00

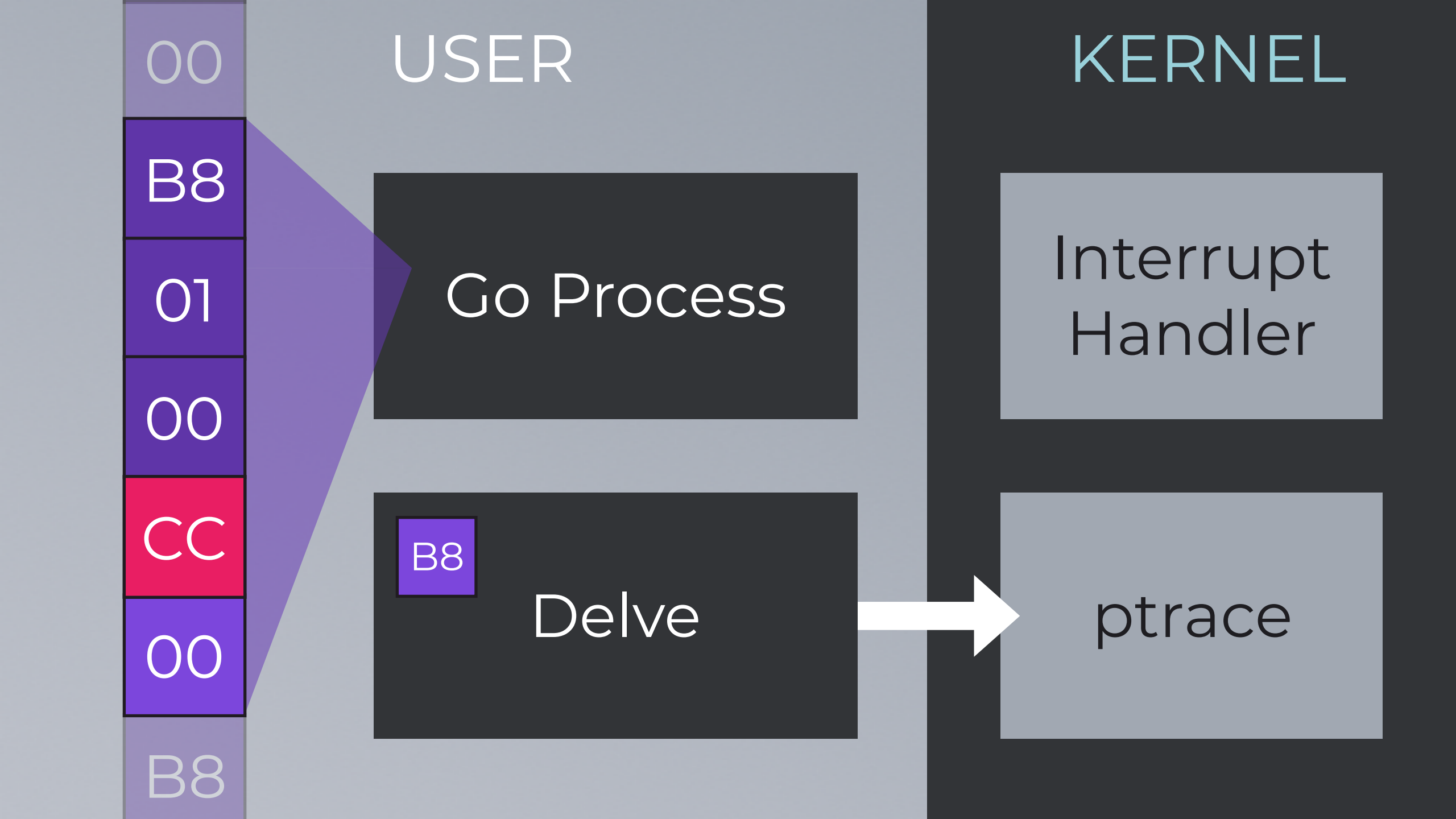

00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
Delve
Interrupt Handler
ptrace


00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
Delve
Interrupt Handler
ptrace

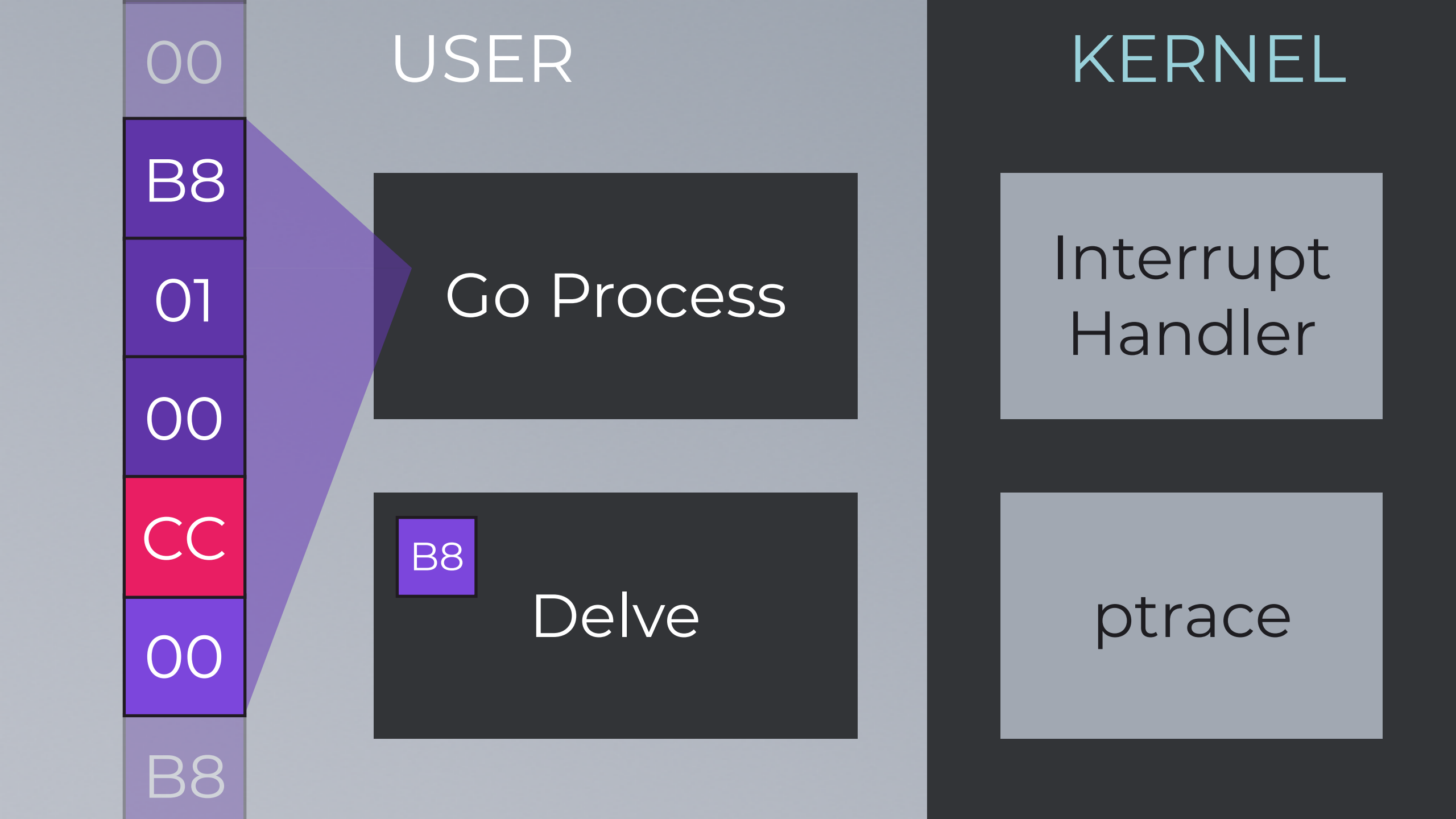
Delve Uses ptrace to Set Breakpoints

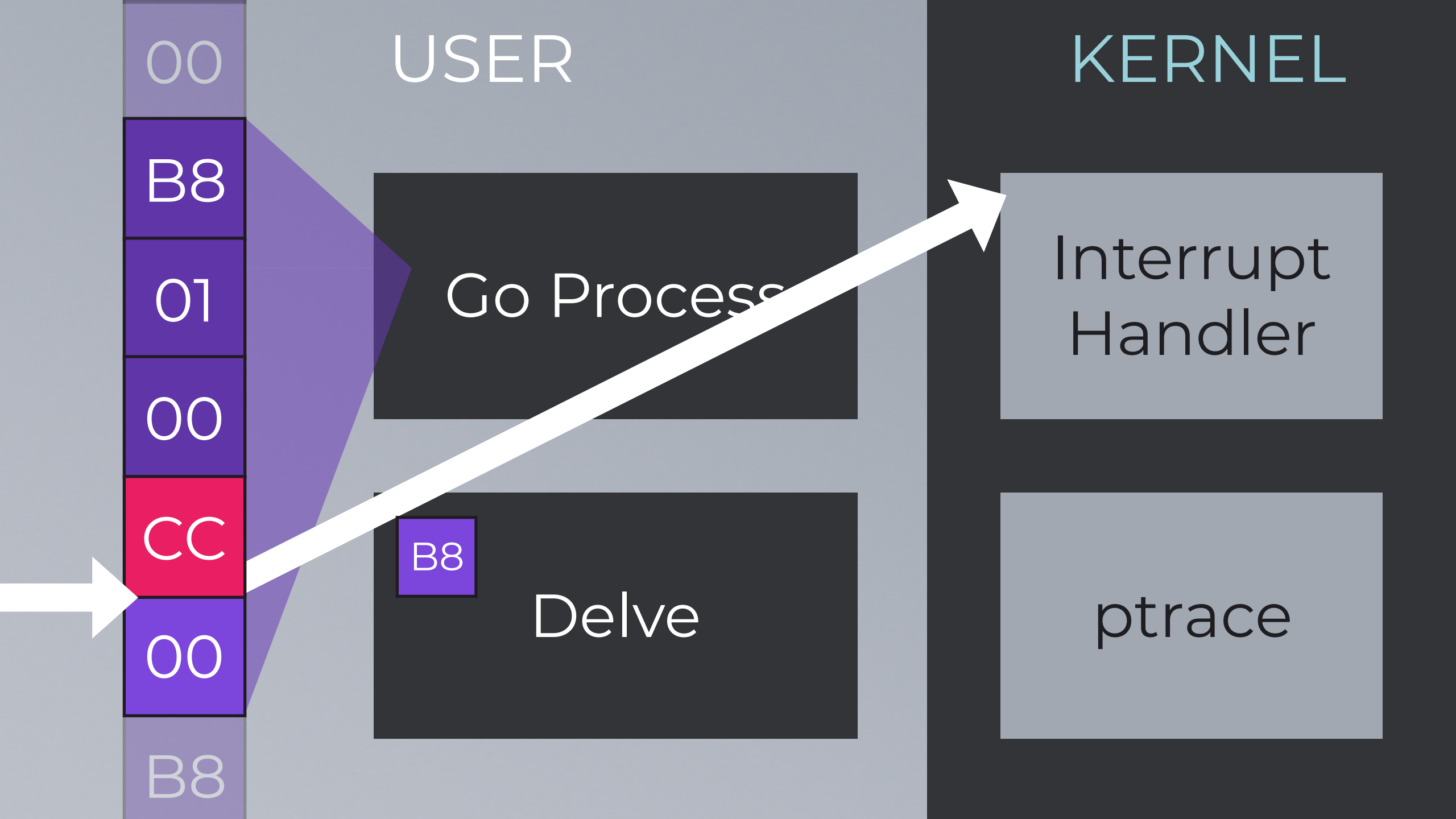

00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
Delve
Interrupt Handler
ptrace

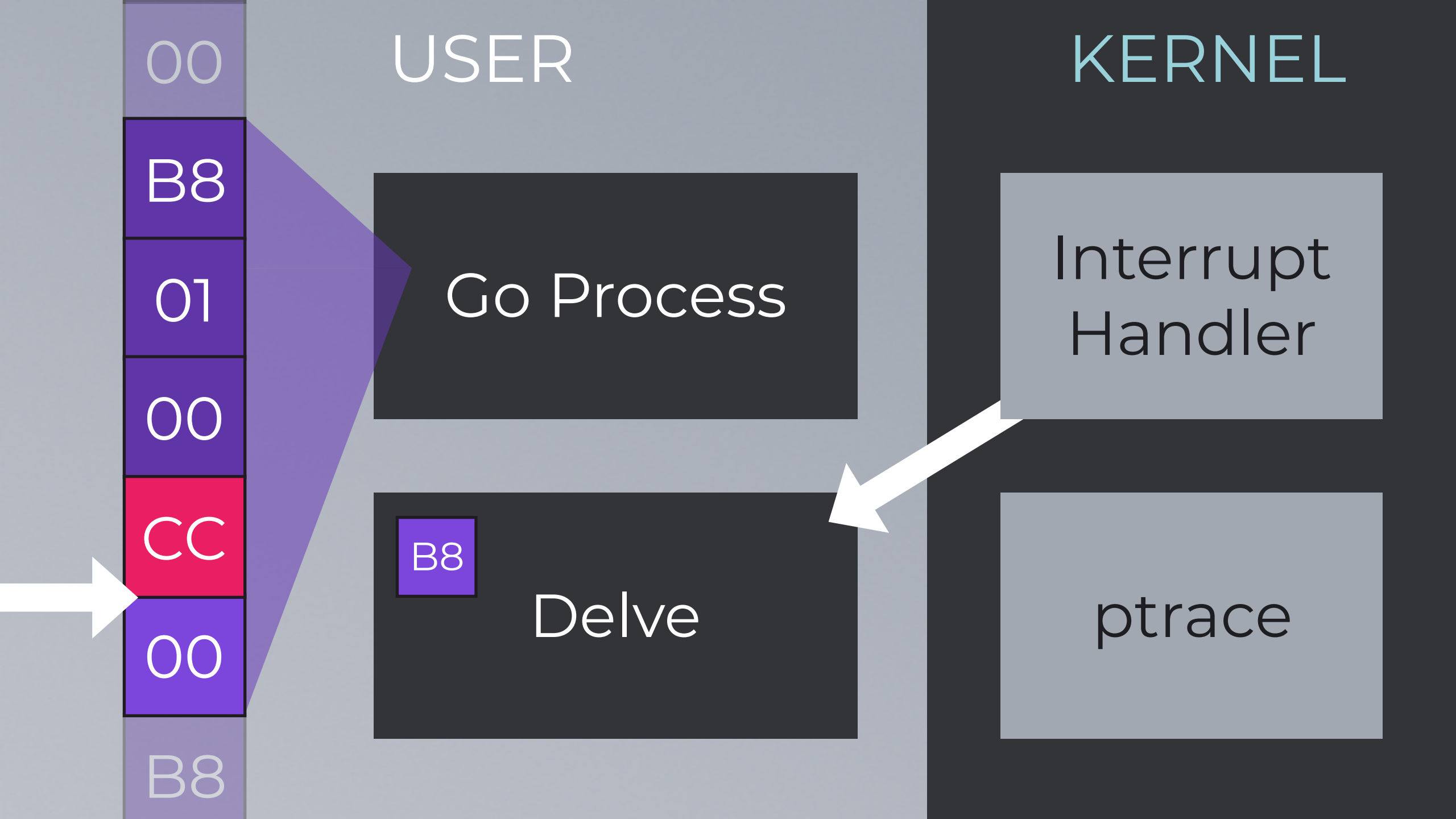

00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
B8
Delve
Interrupt Handler
ptrace

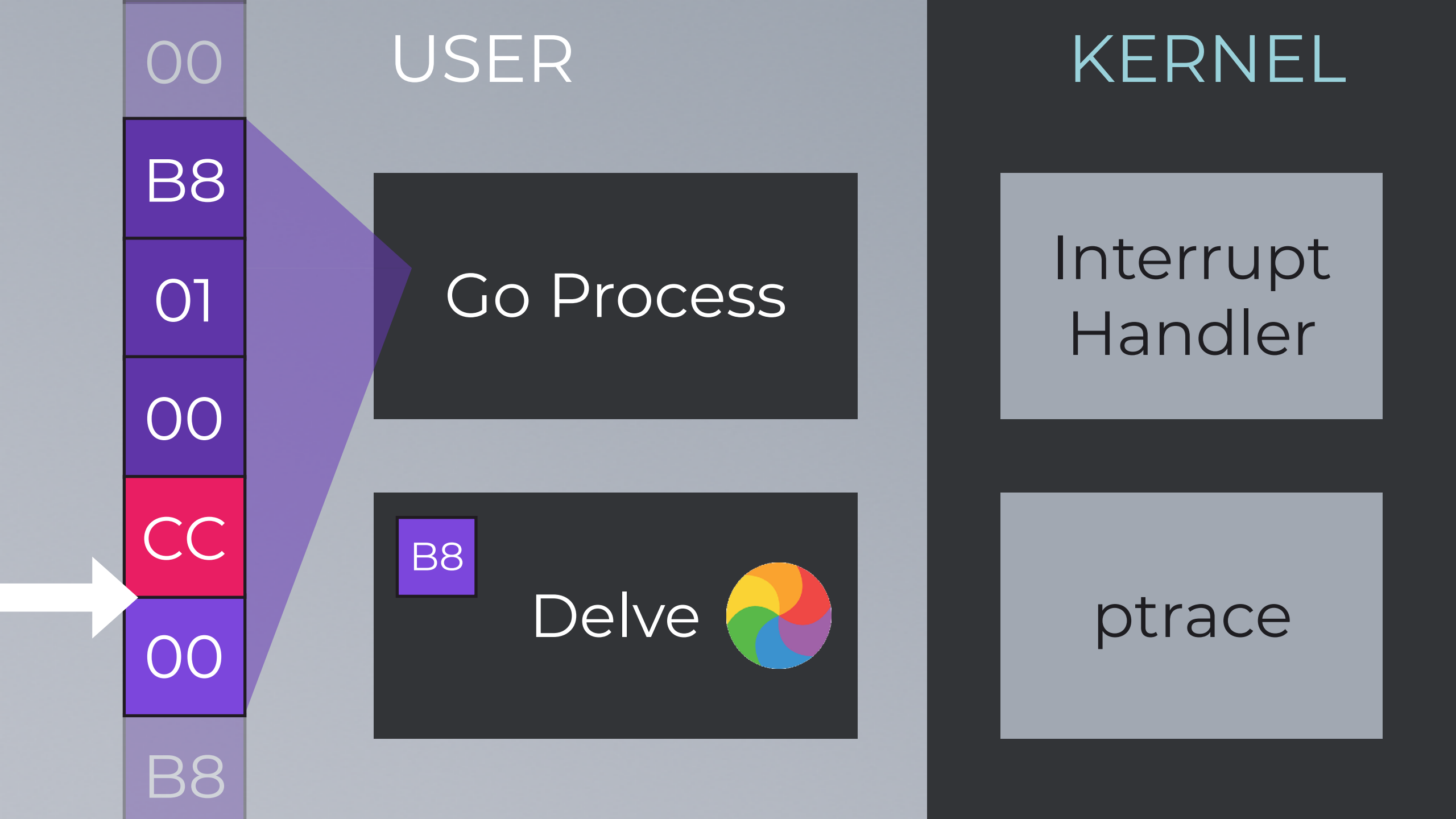

00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
B8
Delve
Interrupt Handler
ptrace

When a Breakpoint is Hit Delve Takes Over

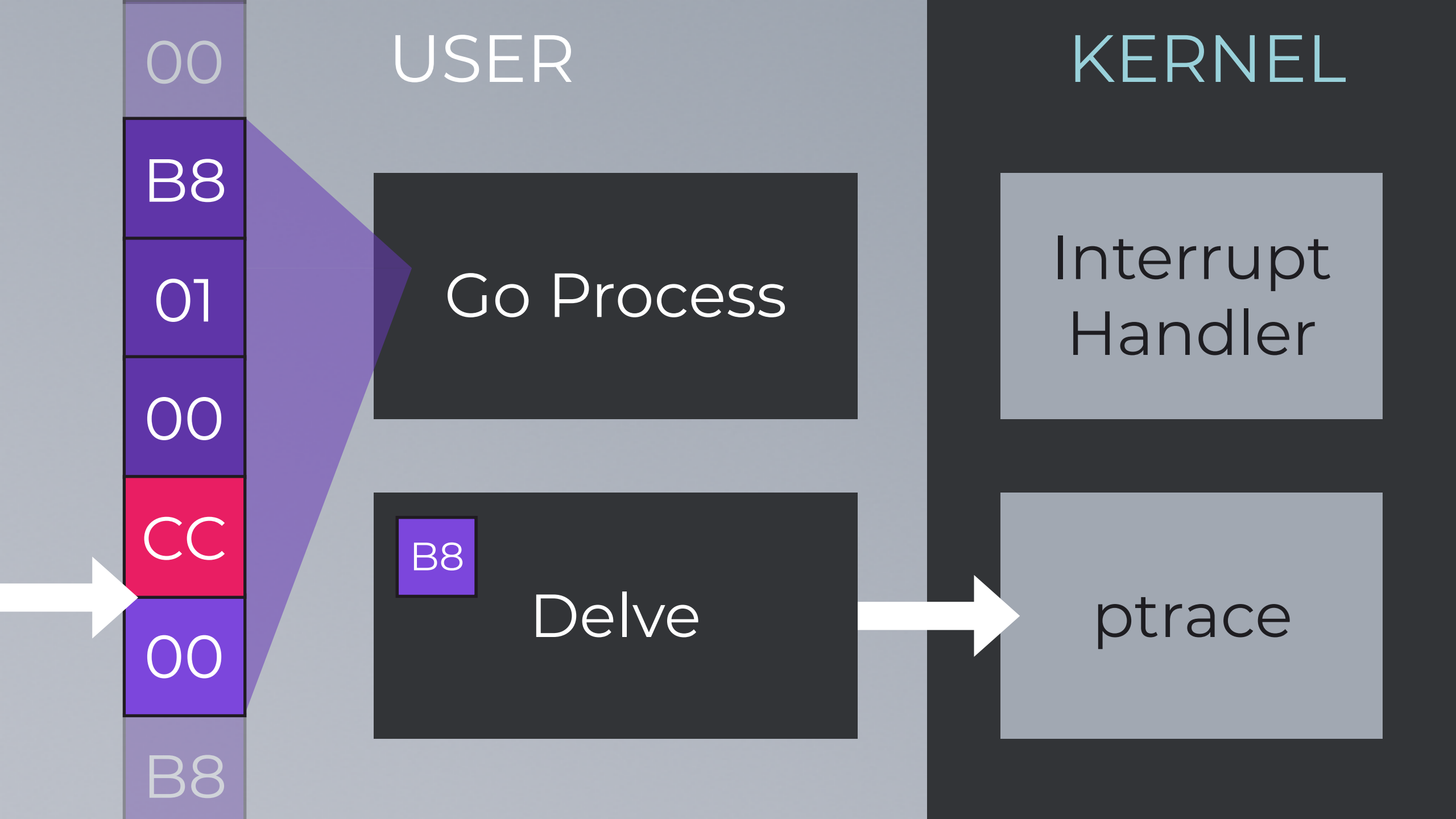

00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
B8
Delve
Interrupt Handler
ptrace

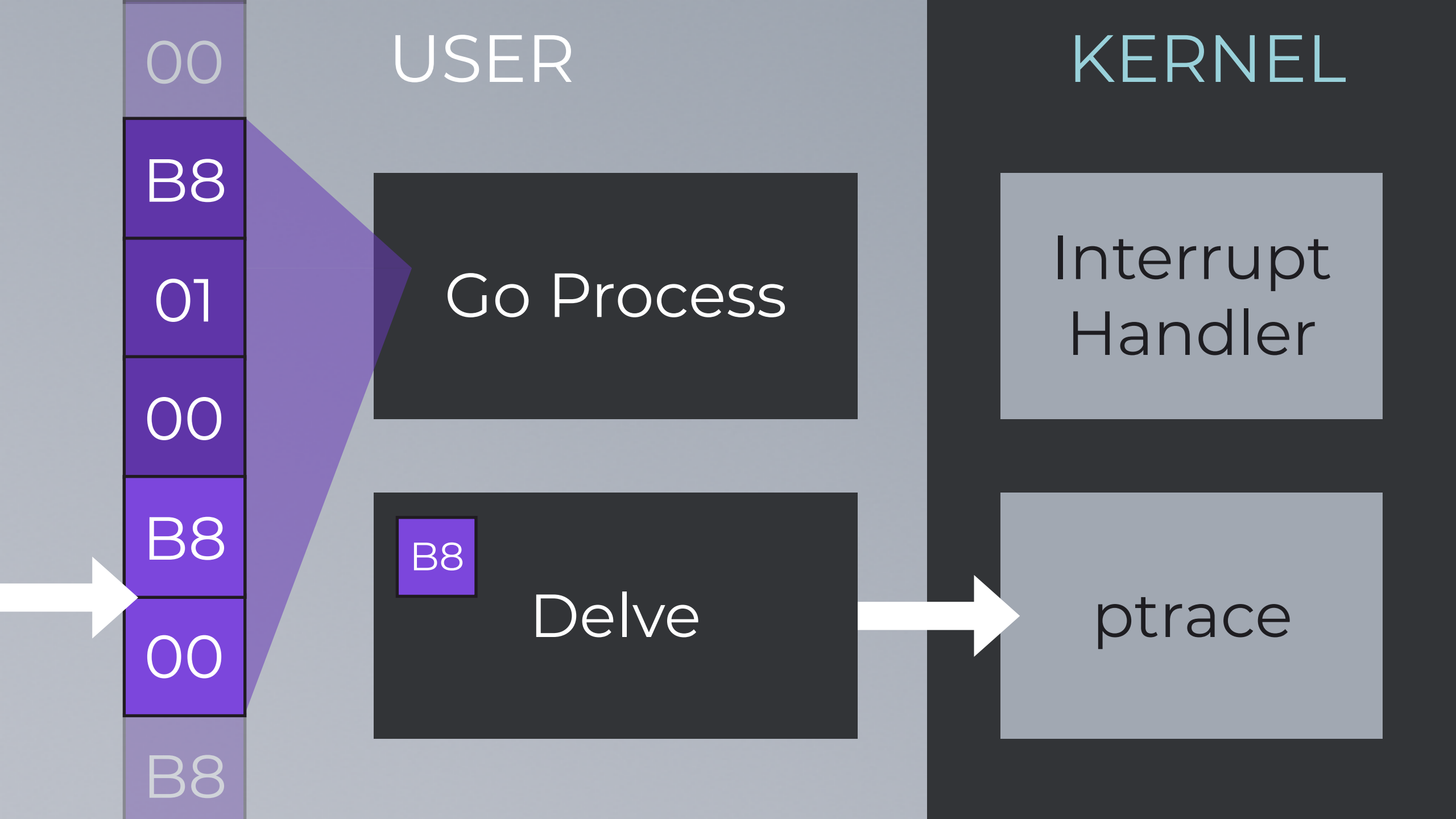

00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
B8
Delve
Interrupt Handler
ptrace

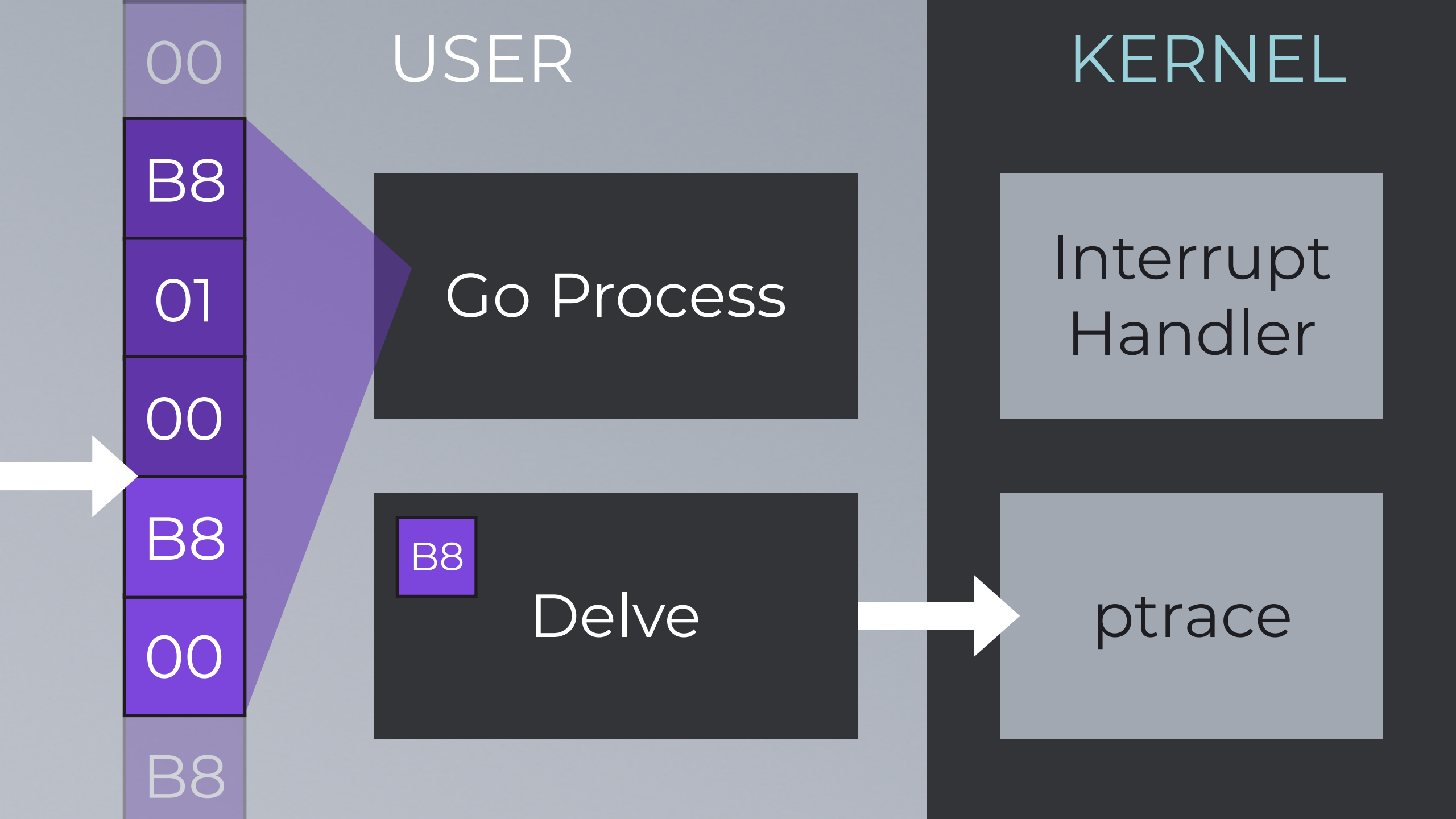

00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
Interrupt Handler
B8
Delve
ptrace

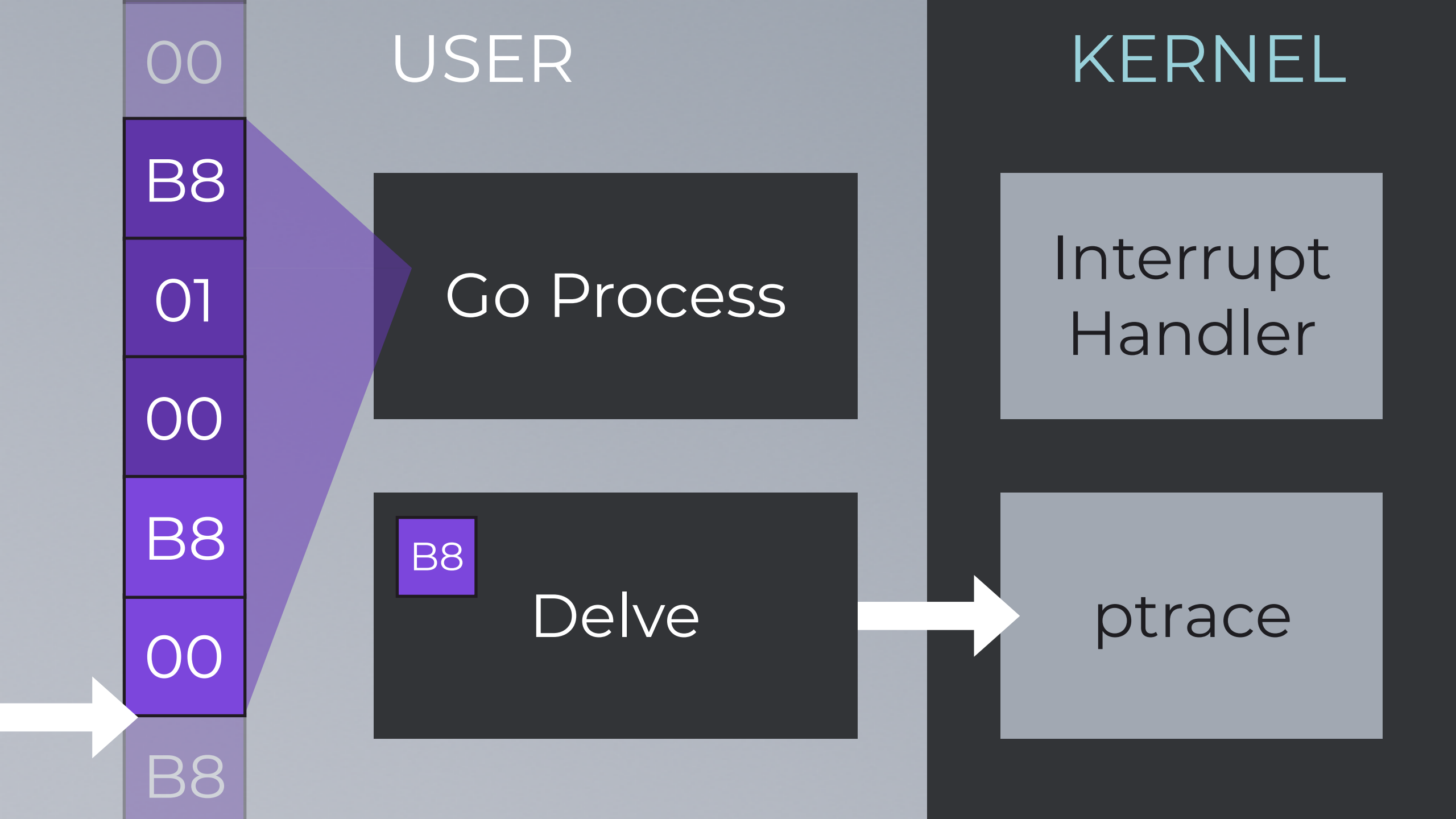

00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
B8
Delve
Interrupt Handler
ptrace

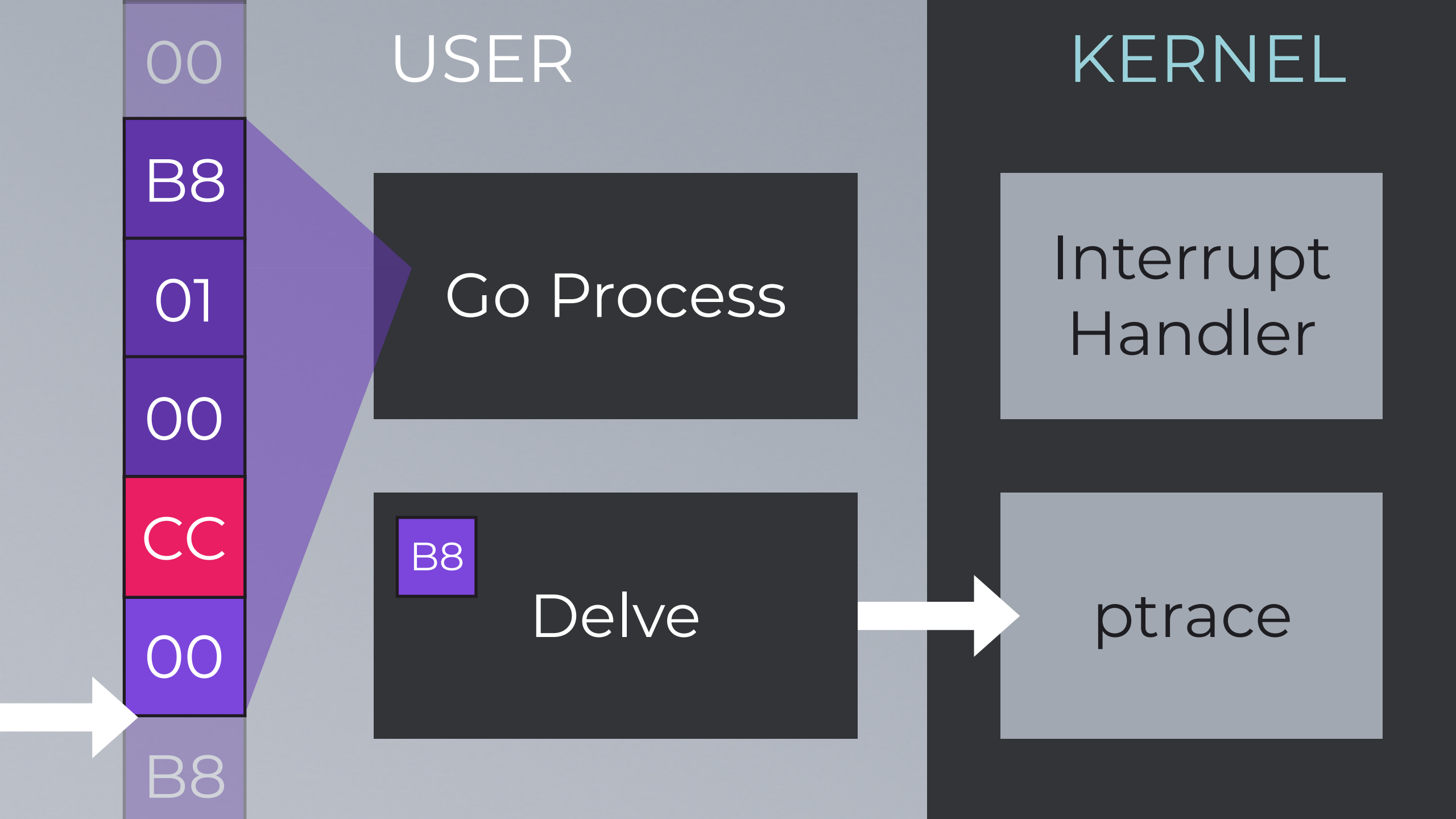

Eventually Delve uses
ptrace to Resume Execution

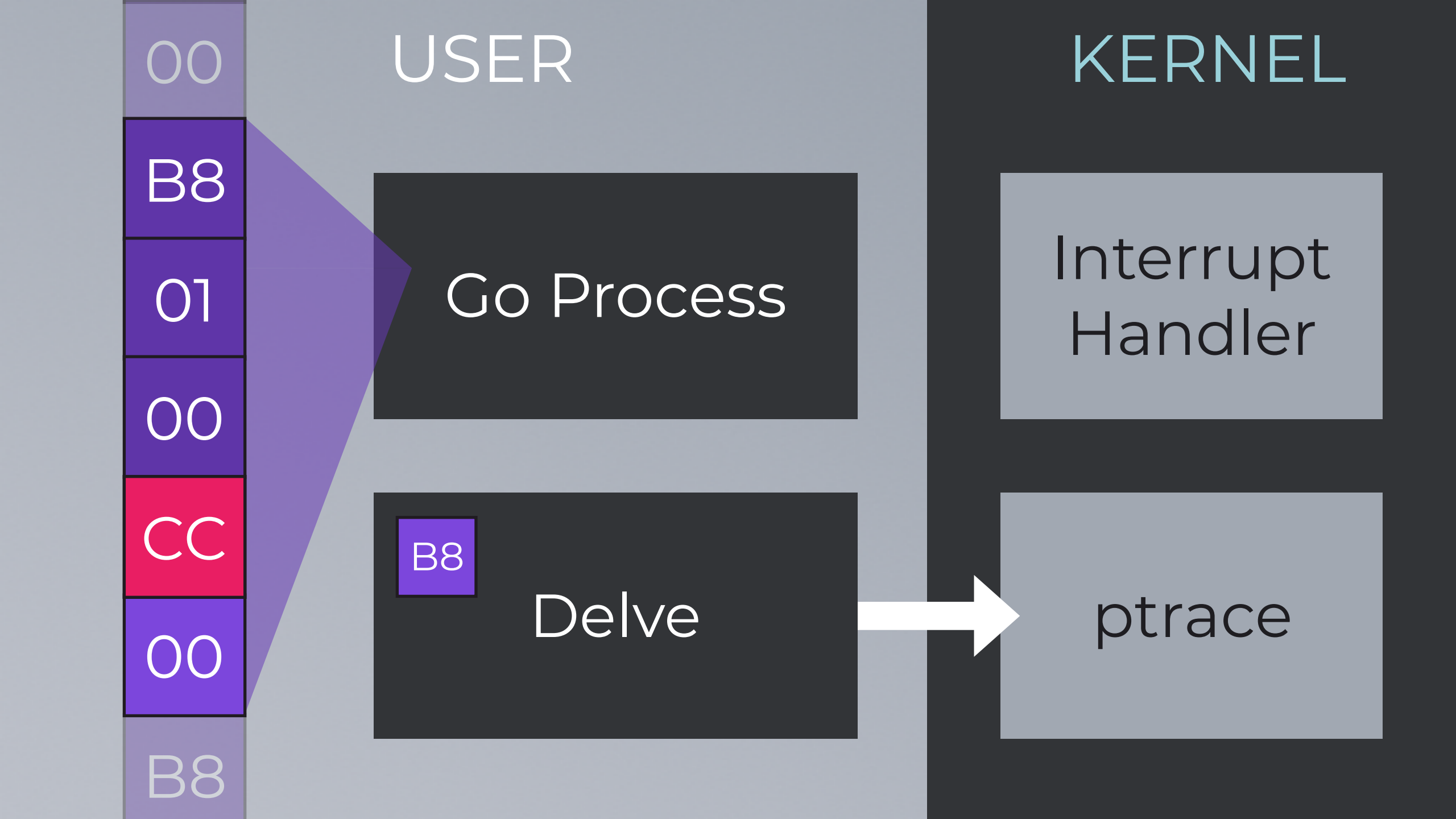

00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
B8
Delve
Interrupt Handler
ptrace


00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
B8
Delve
Interrupt Handler
ptrace

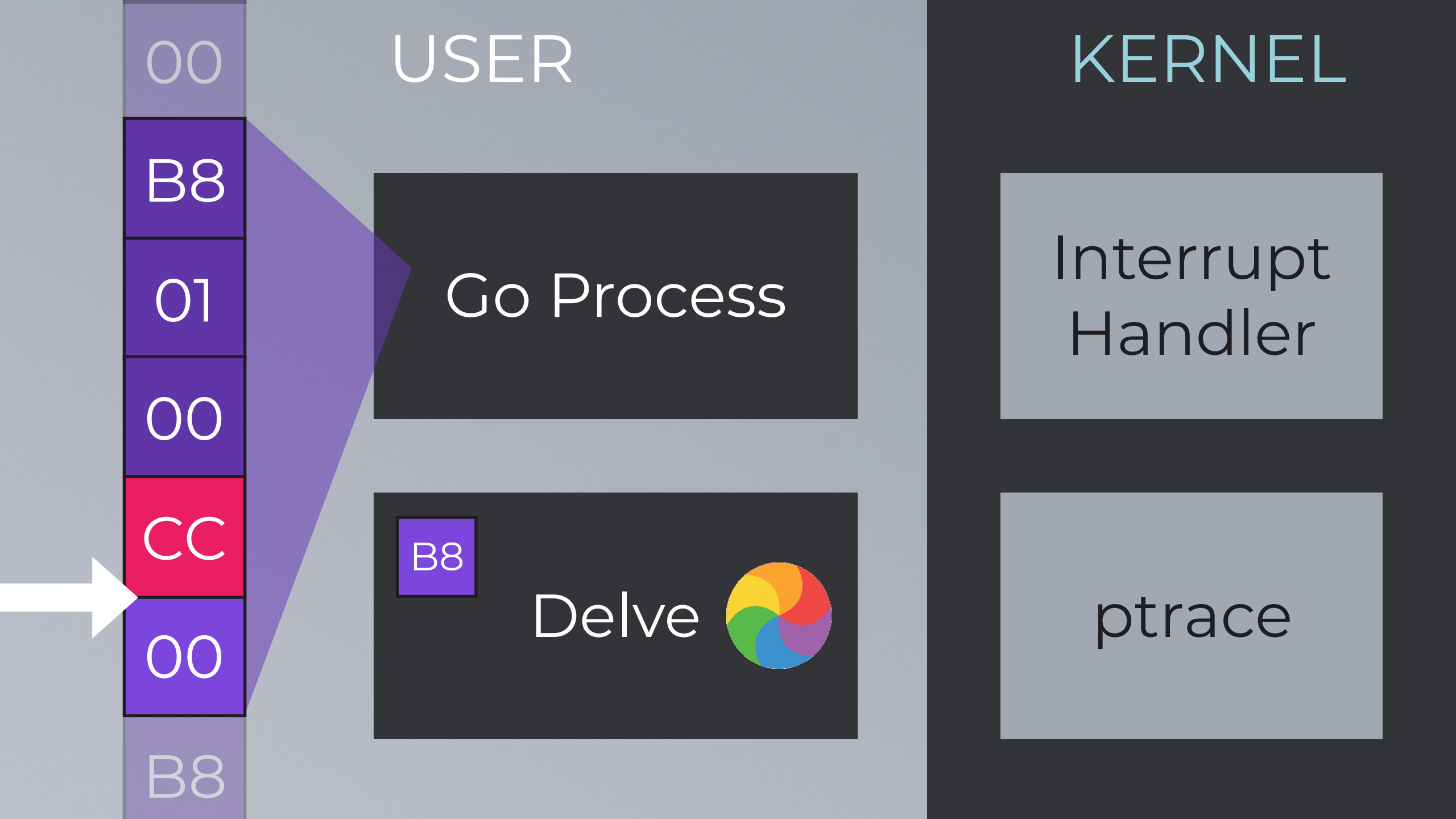

00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
B8
Delve
Interrupt Handler
ptrace


00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
B8
Delve
Interrupt Handler
ptrace


00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
B8
Delve
Interrupt Handler
ptrace


00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
B8
Delve
Interrupt Handler
ptrace

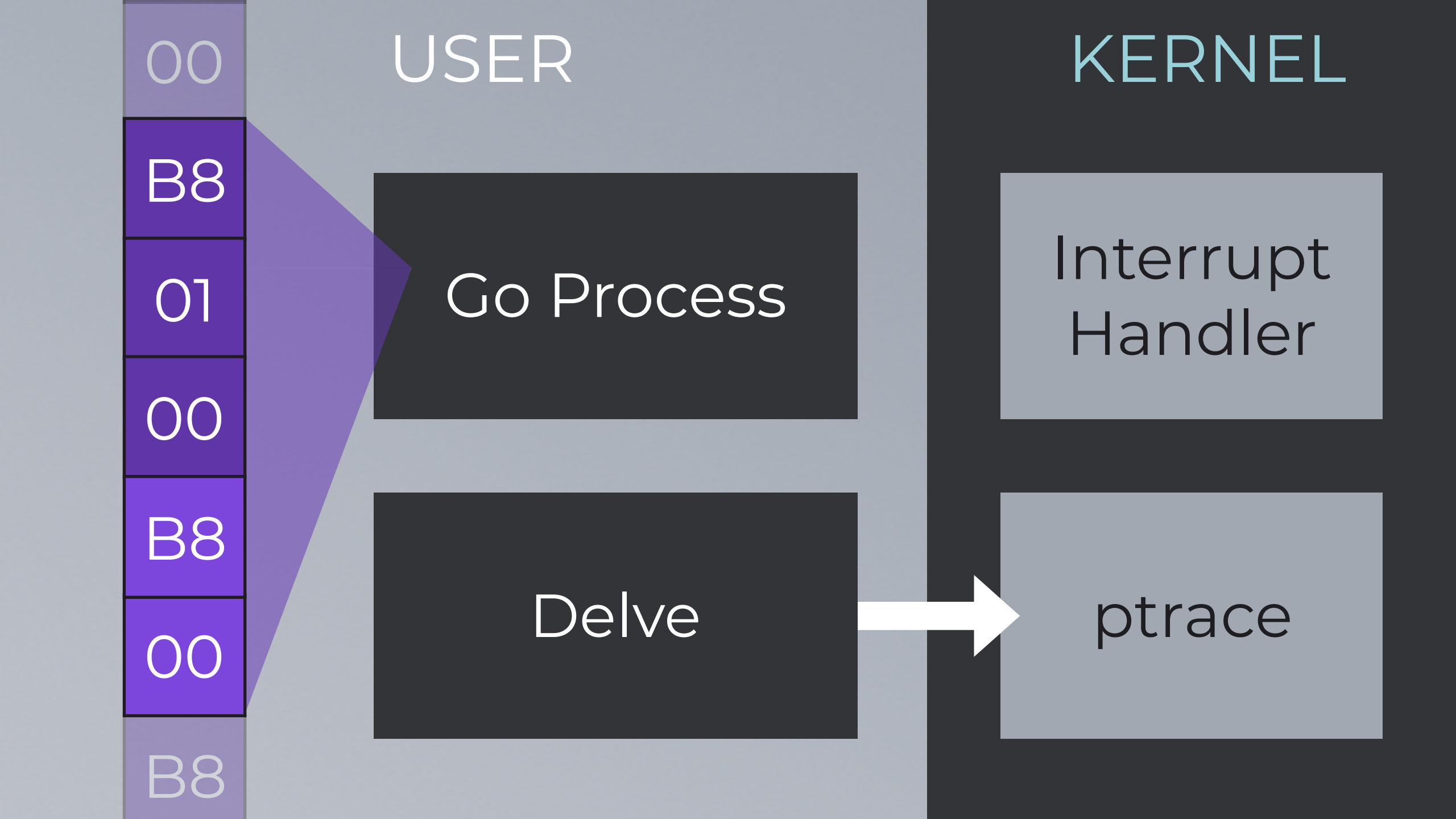
The major problem with this is suspended execution.

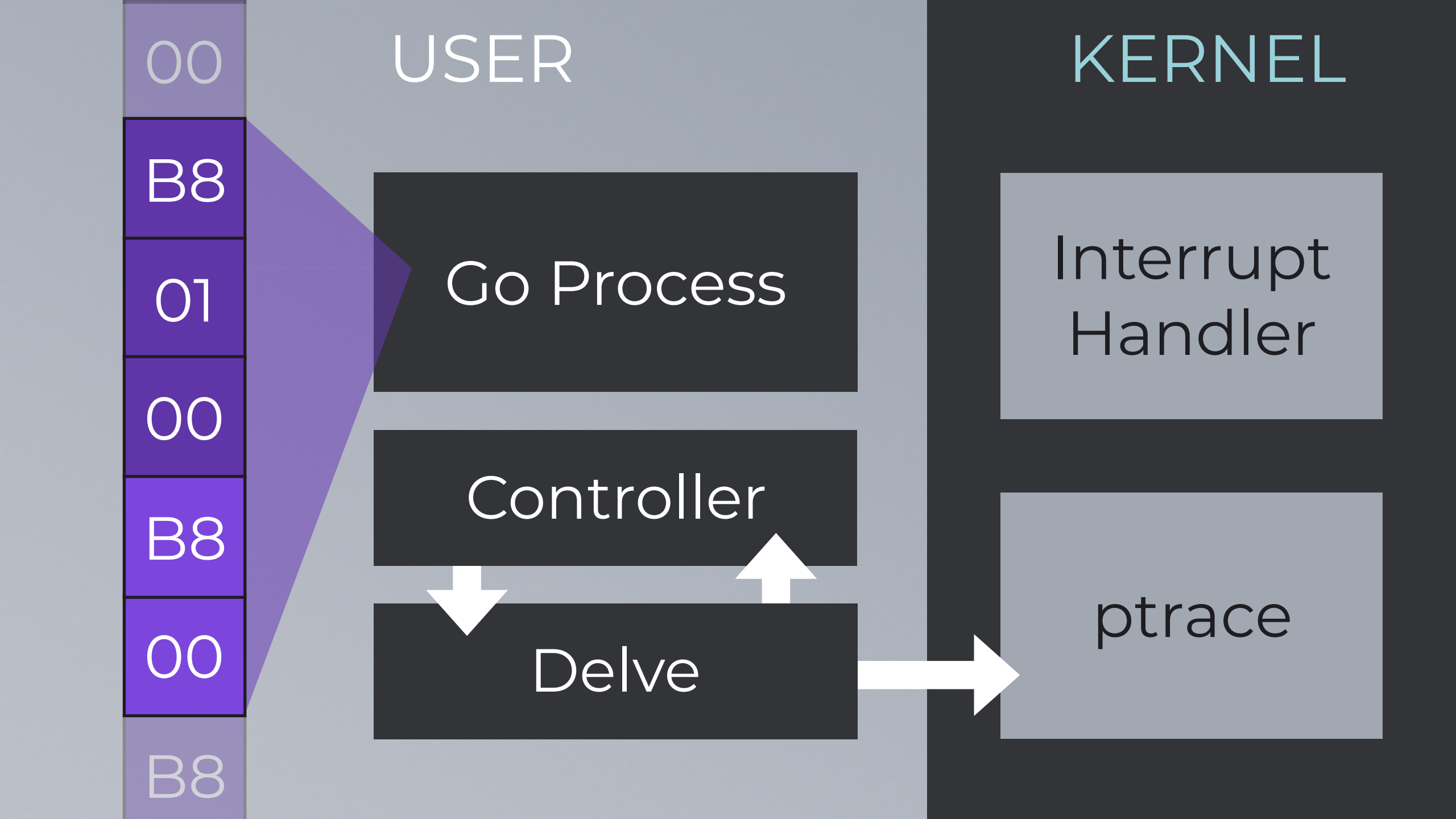

00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
B8
Delve
Interrupt Handler
ptrace

We can minimize the amount of time the debugger is running by automating our interactions with it.

Delve exposes a JSON-RPC API that we can use!

We can write a program that uses delve's API to add the instrumentation we want in an automated way!


00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
Delve
Interrupt Handler
ptrace


00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
Controller
Delve
Interrupt Handler
ptrace

Let’s first create a test program that requires some dynamic instrumentation.

```
var counter int64

func doWork() {
   counter++
}

func main() {
   fmt.Println("This is a blackbox. Read my counter.")
   for {
      doWork()
   }
}
```

```
$ /tmp/counter
```

```
dlv := exec.Command(
   "dlv",
   "attach",
   strconv.Itoa(pid),
   "--headless",
   "--accept-multiclient",
   "--api-version=2",
   "--listen="+addr,
)
dlv.Start()
waitToBeUp(addr)
```

```
func waitToBeUp(addr string) {
    done := time.After(5 * time.Second)
    for {
        conn, err := net.DialTimeout("tcp", addr, 2*time.Second)
        if err == nil {
            conn.Close()
            return
        }
        select {
        case <-done:
            log.Fatalf("Stuck waiting for %q to be up: %s", addr, err)
        default:
        }
    }
}
```

```
client := rpc2.NewClient(addr)
client.Continue()
defer func() {
   client.Halt()
   client.Detach(false)
}()
```

```
locations, _ := client.FindLocation(api.EvalScope{
   GoroutineID: -1,
}, "main.doWork")
pc := locations[0].PC
```

```
bp := &api.Breakpoint{
   Name:      "myAwesomeBreakpoint",
   Addr:      pc,
   Variables: []string{"counter"},
}
```

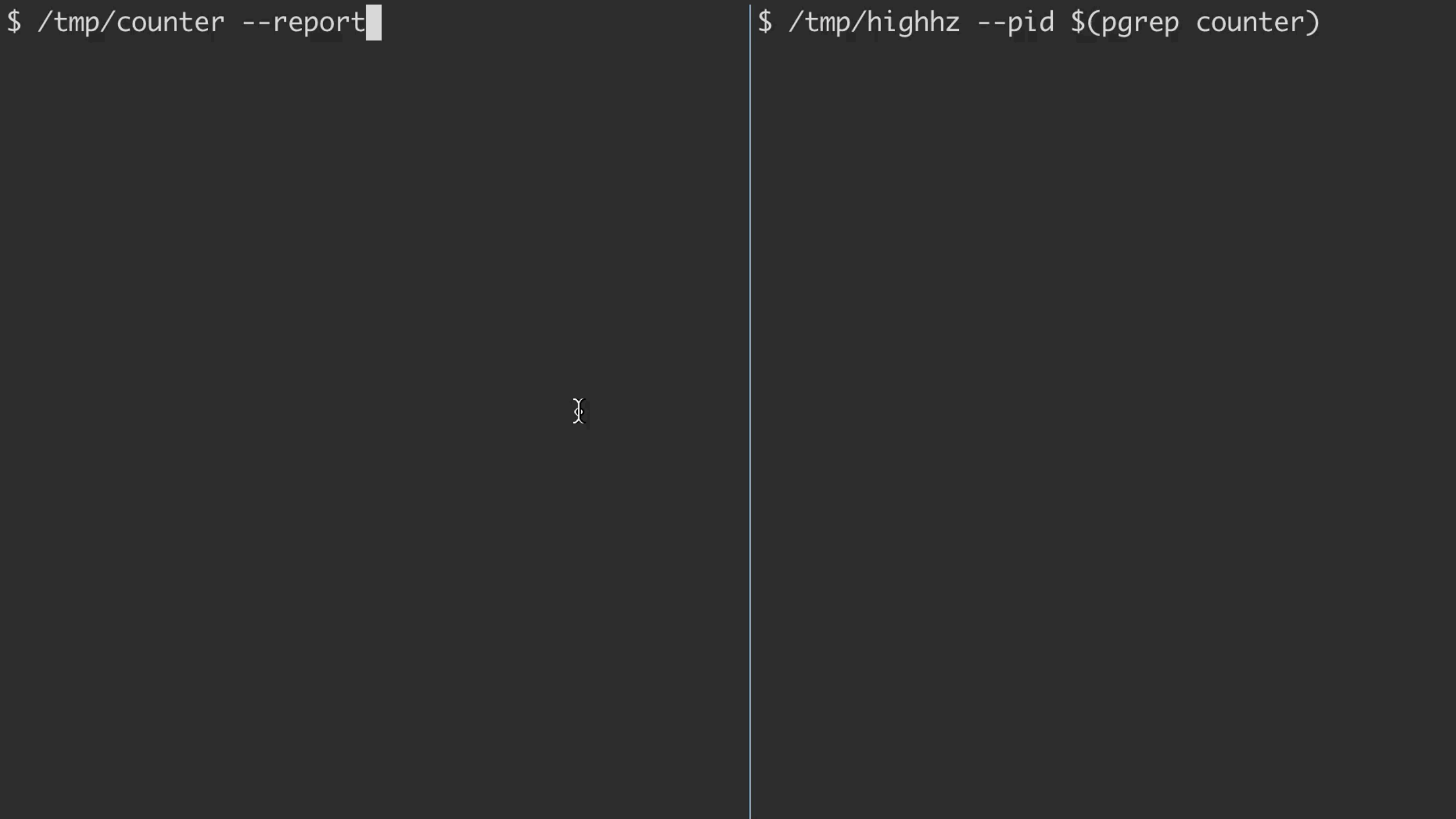
```
for {
    client.Halt()
    client.CreateBreakpoint(bp)
    state := <-client.Continue()
    client.ClearBreakpointByName(bp.Name)
    client.Continue()

    counter := state.CurrentThread.BreakpointInfo.Variables[0].Value
    fmt.Printf("counter: %s", counter)

    time.Sleep(time.Second)
}
```

```
$ /tmp/counter --report
```

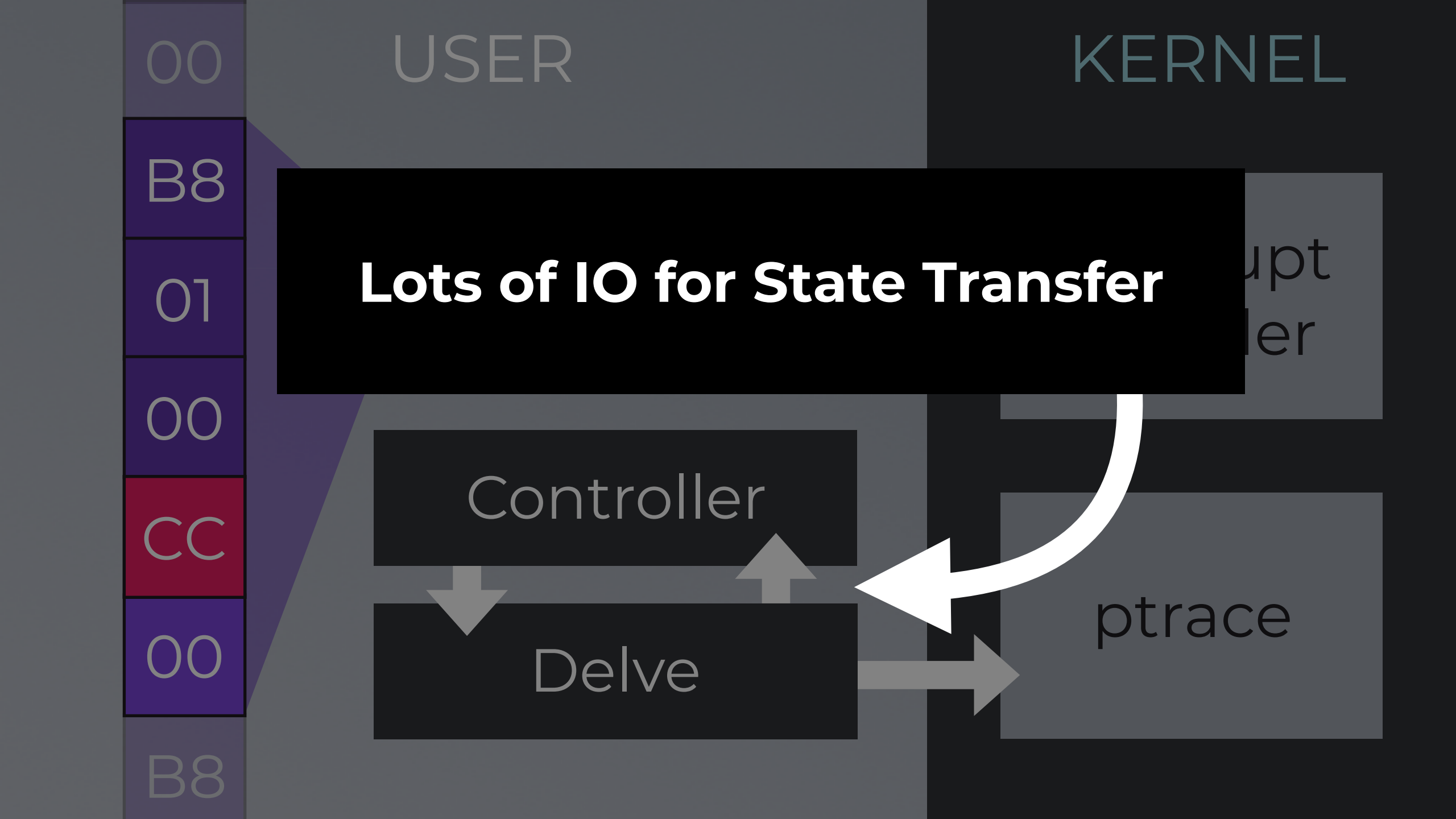
```
$ /tmp/lowhz --pid $(pgrep counter)
```

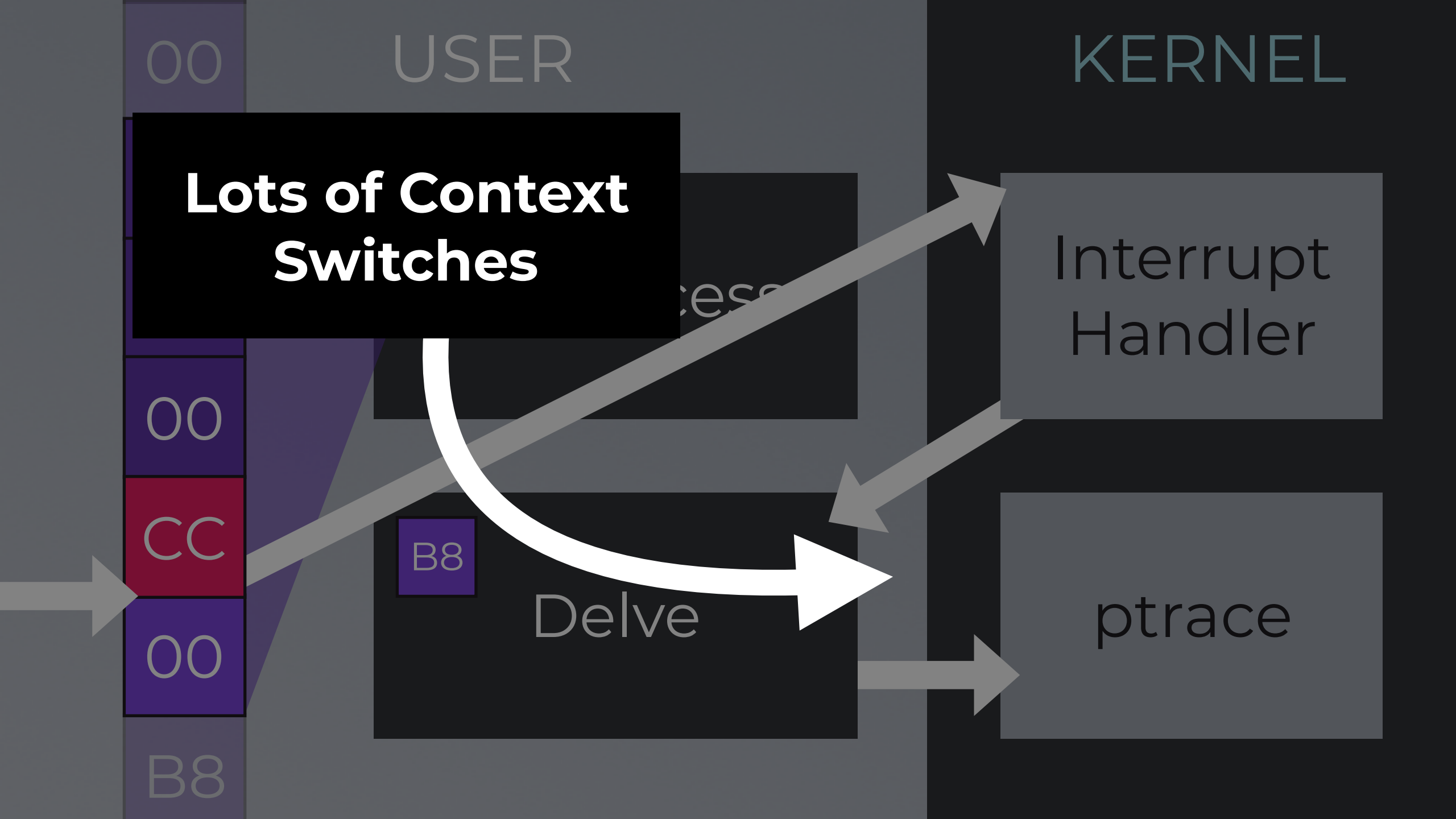
What happens if we intercept every call to doWork?

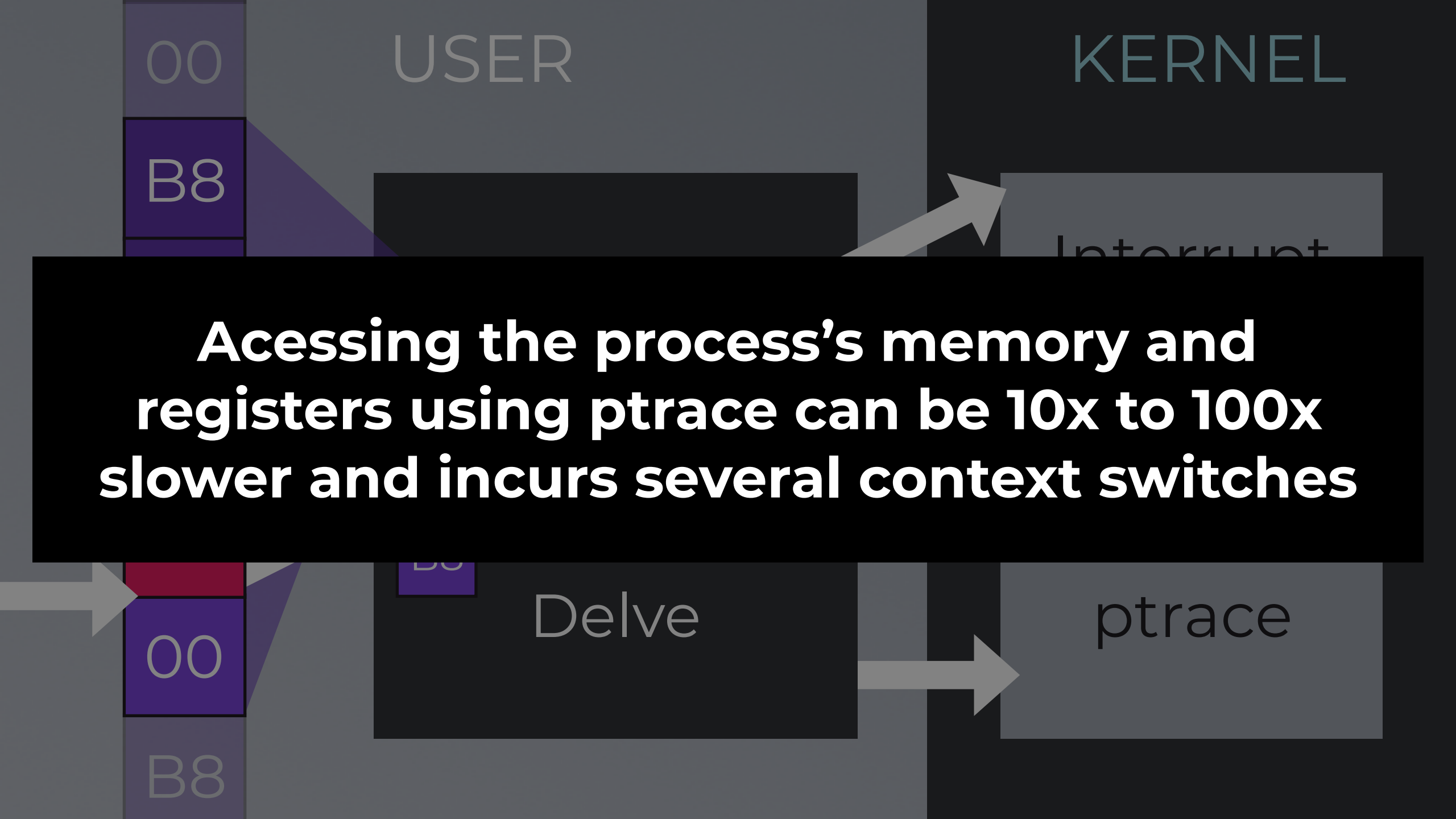
```
var sharedCounter int64
go func() {
    for {
        fmt.Printf("counter: %d", atomic.LoadInt64(&sharedCounter))
        time.Sleep(time.Second)
    }
}()

client.CreateBreakpoint(bp)
stateCh := client.Continue()

for {
    state := <-stateCh
    stateCh = client.Continue()

    counterStr := state.CurrentThread.BreakpointInfo.Variables[0].Value
    counter, _ := strconv.ParseInt(counterStr, 10, 64)
    atomic.StoreInt64(&sharedCounter, counter)
}
```


$ /tmp/counter --report
$ /tmp/highhz --pid $(pgrep counter)

Why are High Frequency Breakpoints So Bad?

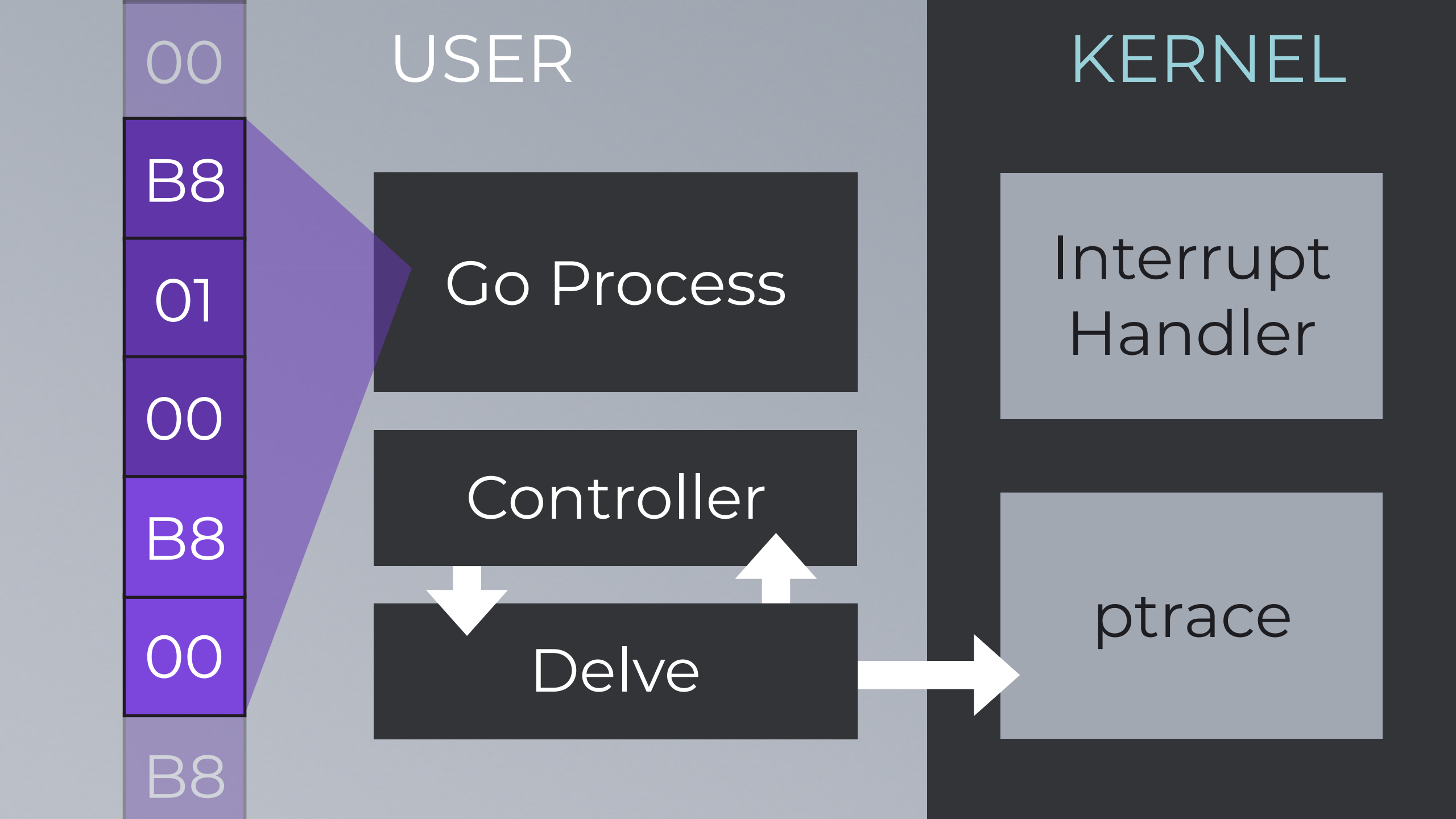

00
B8
01
00
CC
00
B8
USER
KERNEL
Lots of IO for State Transfer
Controller
Delve
ptrace

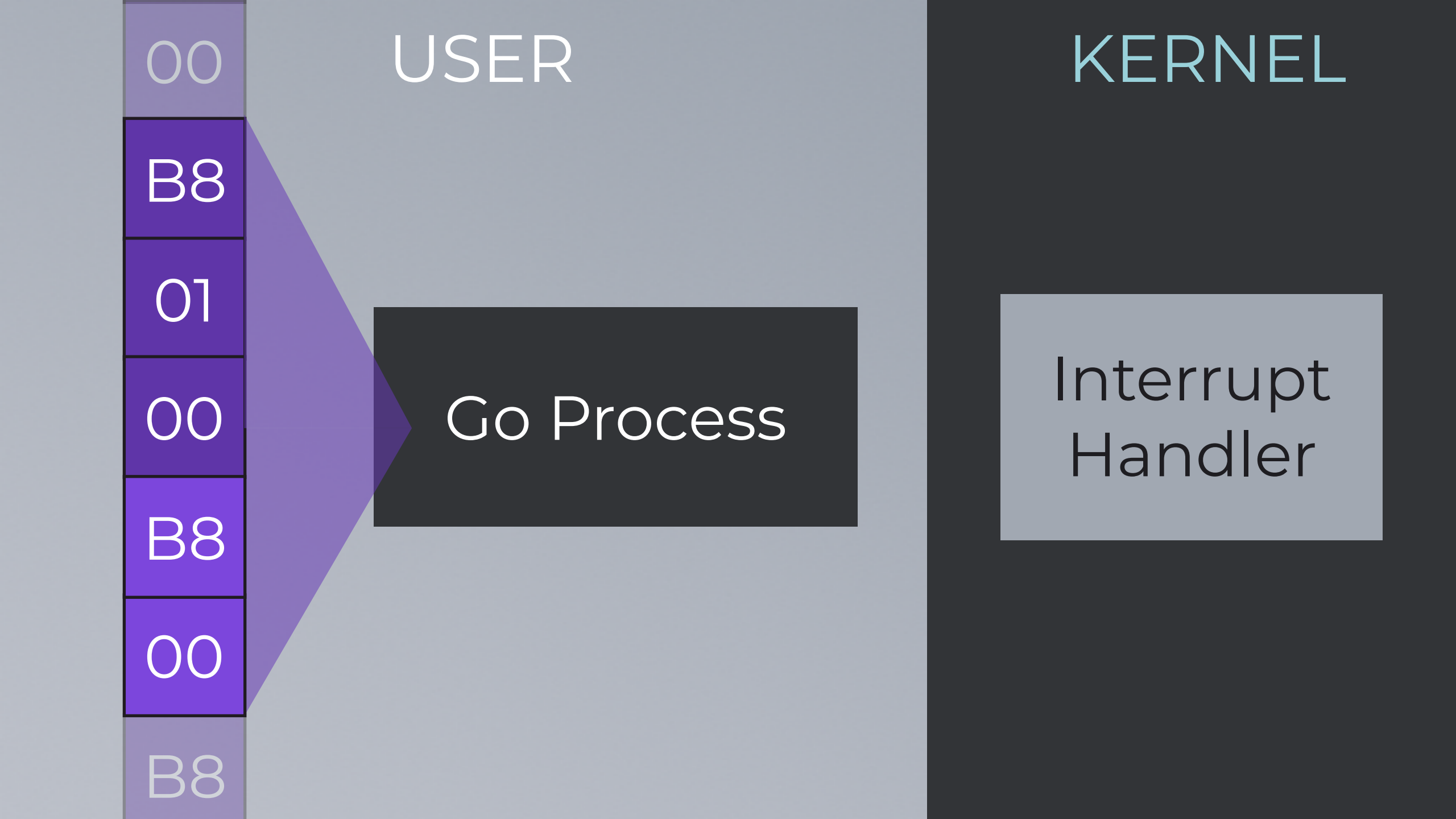

00
USER
KERNEL
Lots of Context Switches
Interrupt Handler
00
CC
B8
Delve
ptrace
00
B8

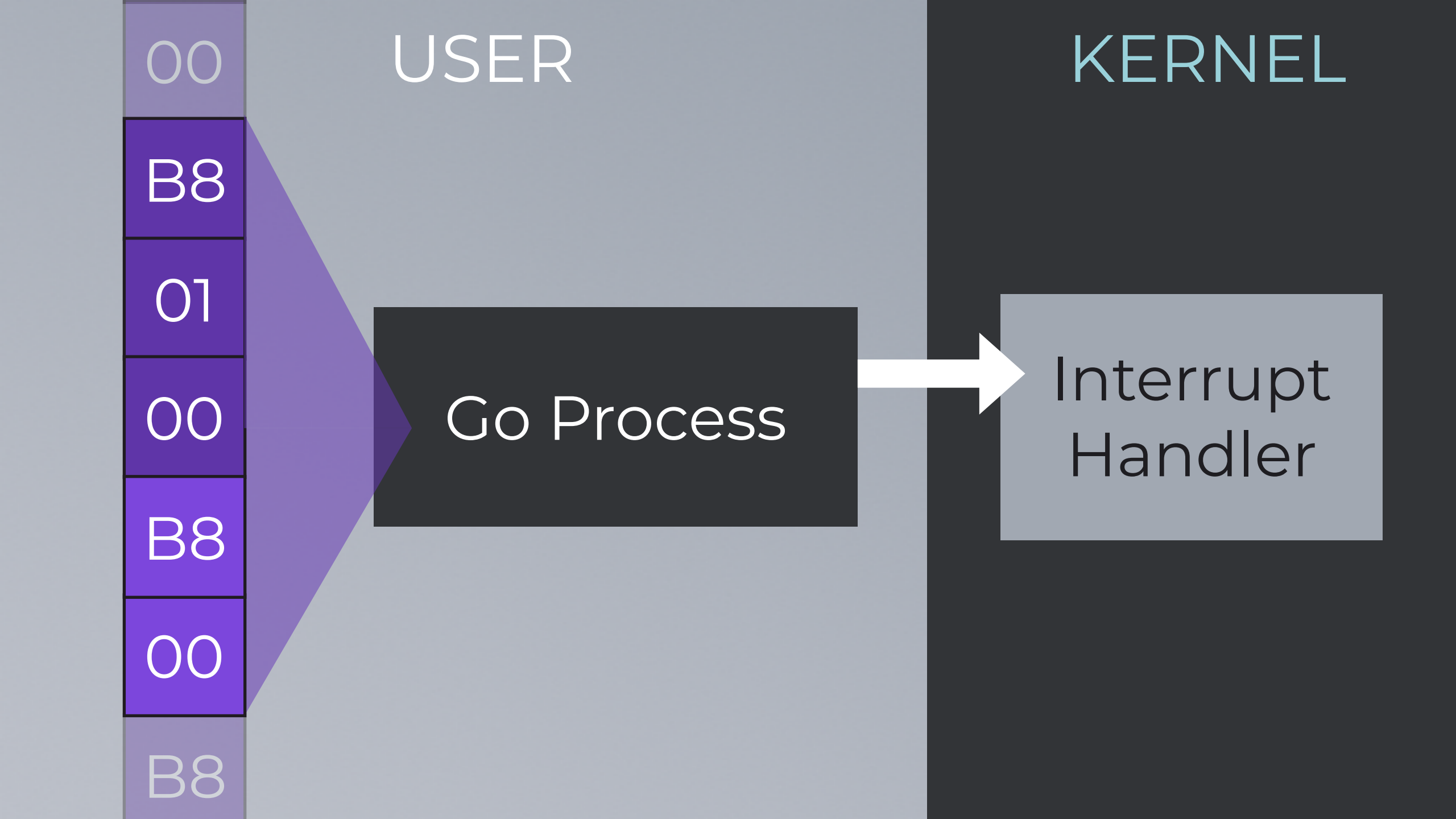


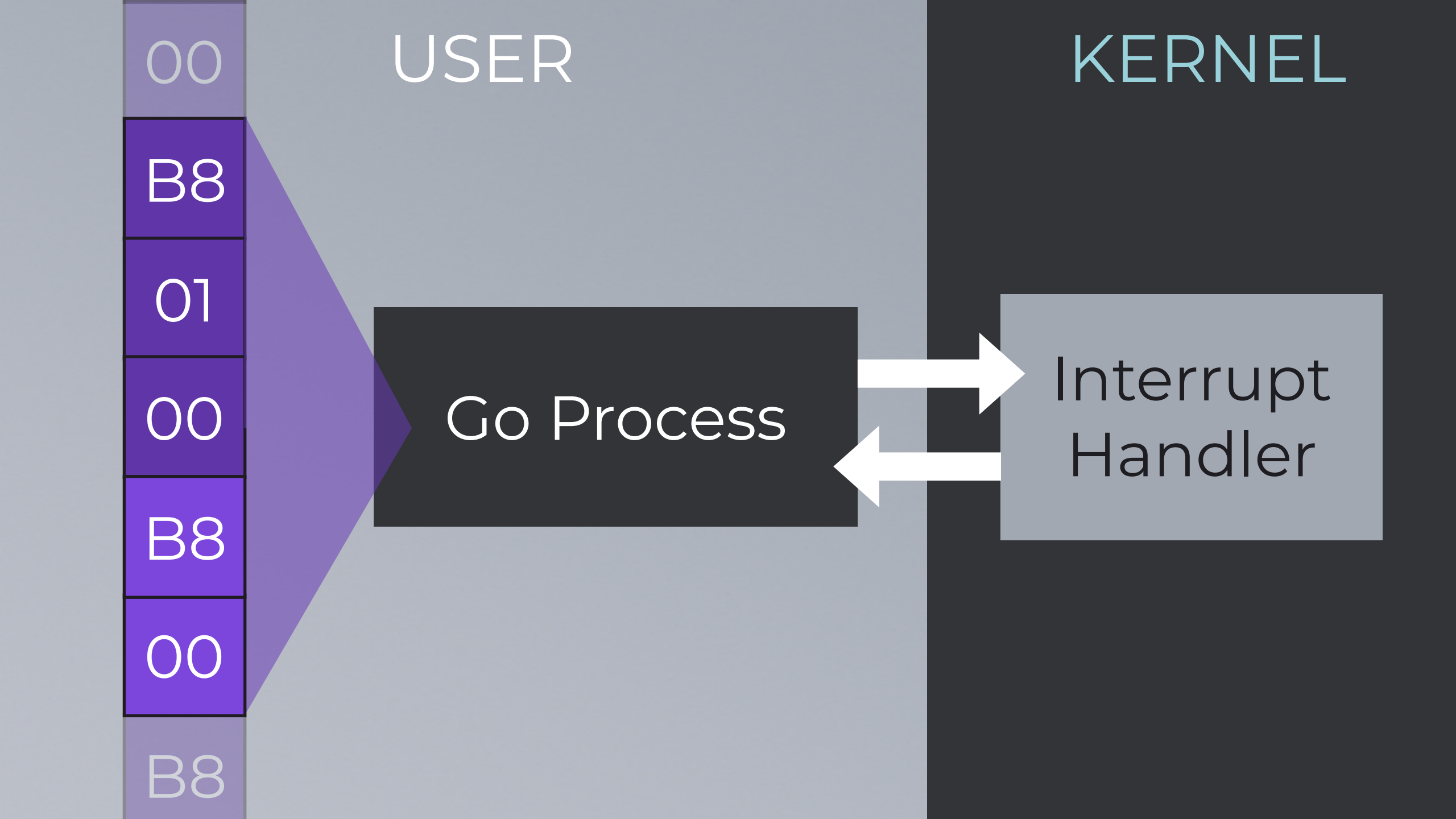
Acessing the process's memory and registers using ptrace can be 10x to 100x slower and incurs several context switches

Can We Do Better?


00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
Controller
Delve
Interrupt Handler
ptrace


00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
Interrupt
Handler


00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
Interrupt
Handler

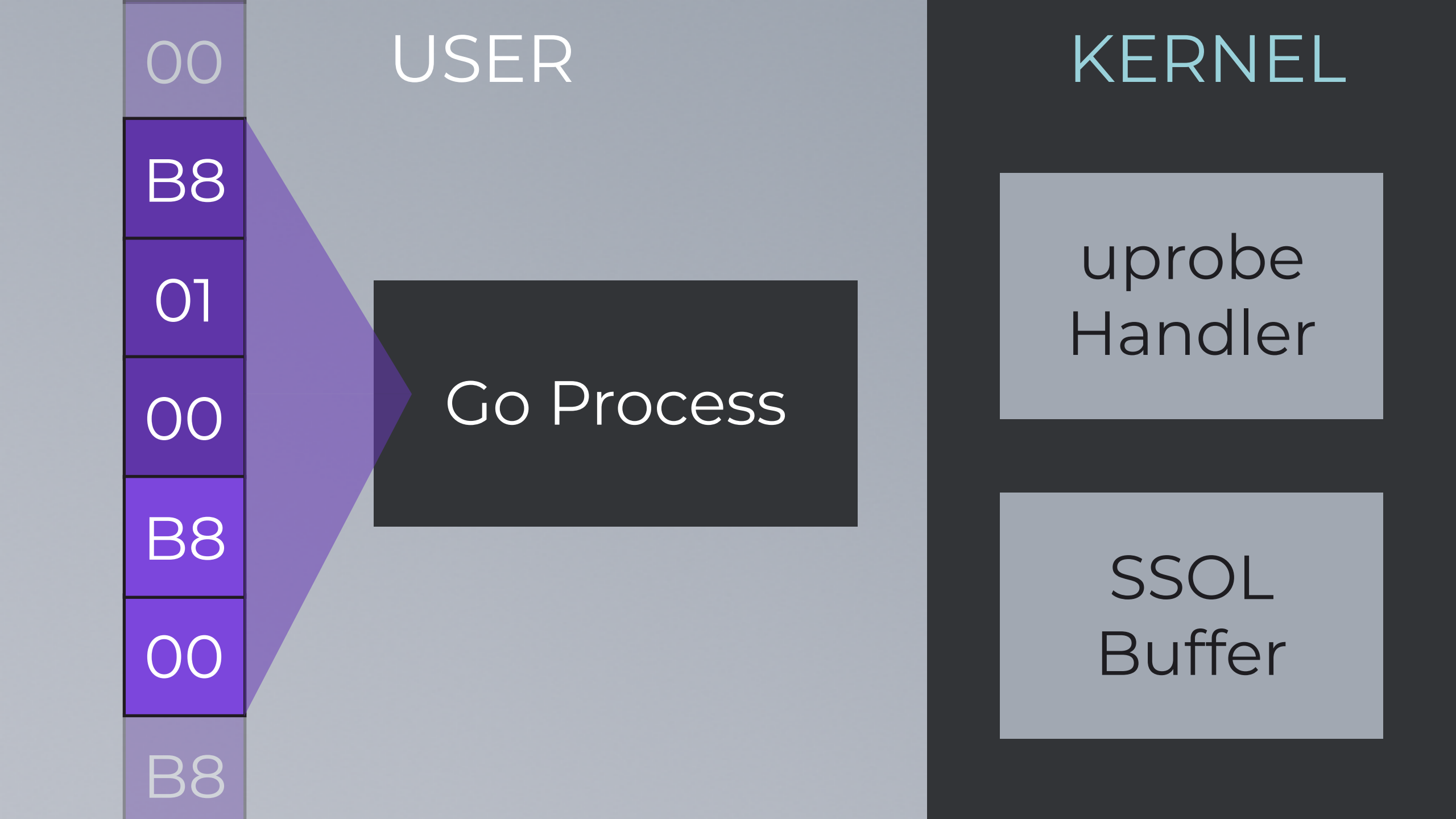

00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
Interrupt
Handler

uprobes+BPF

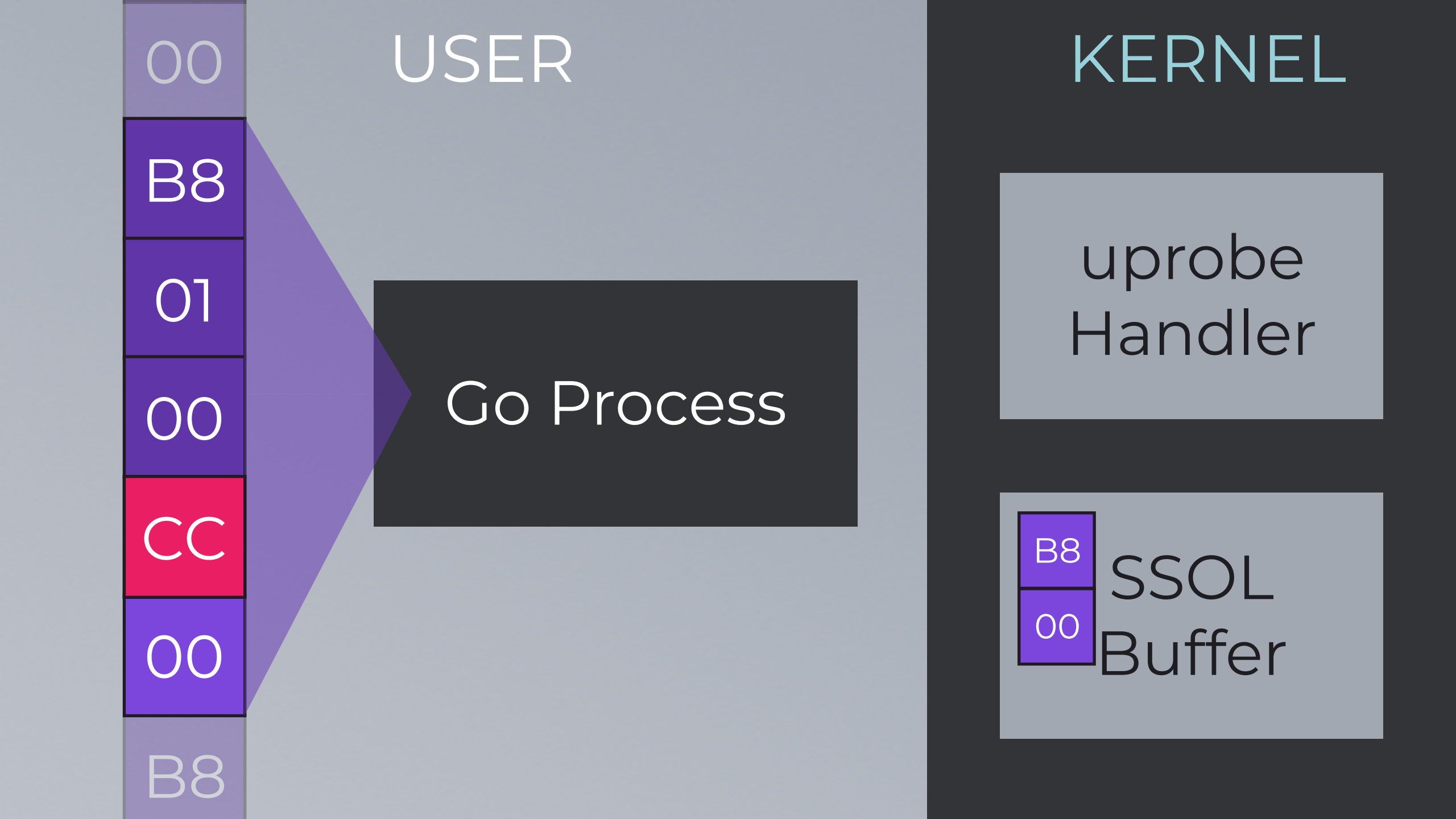
uprobes

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

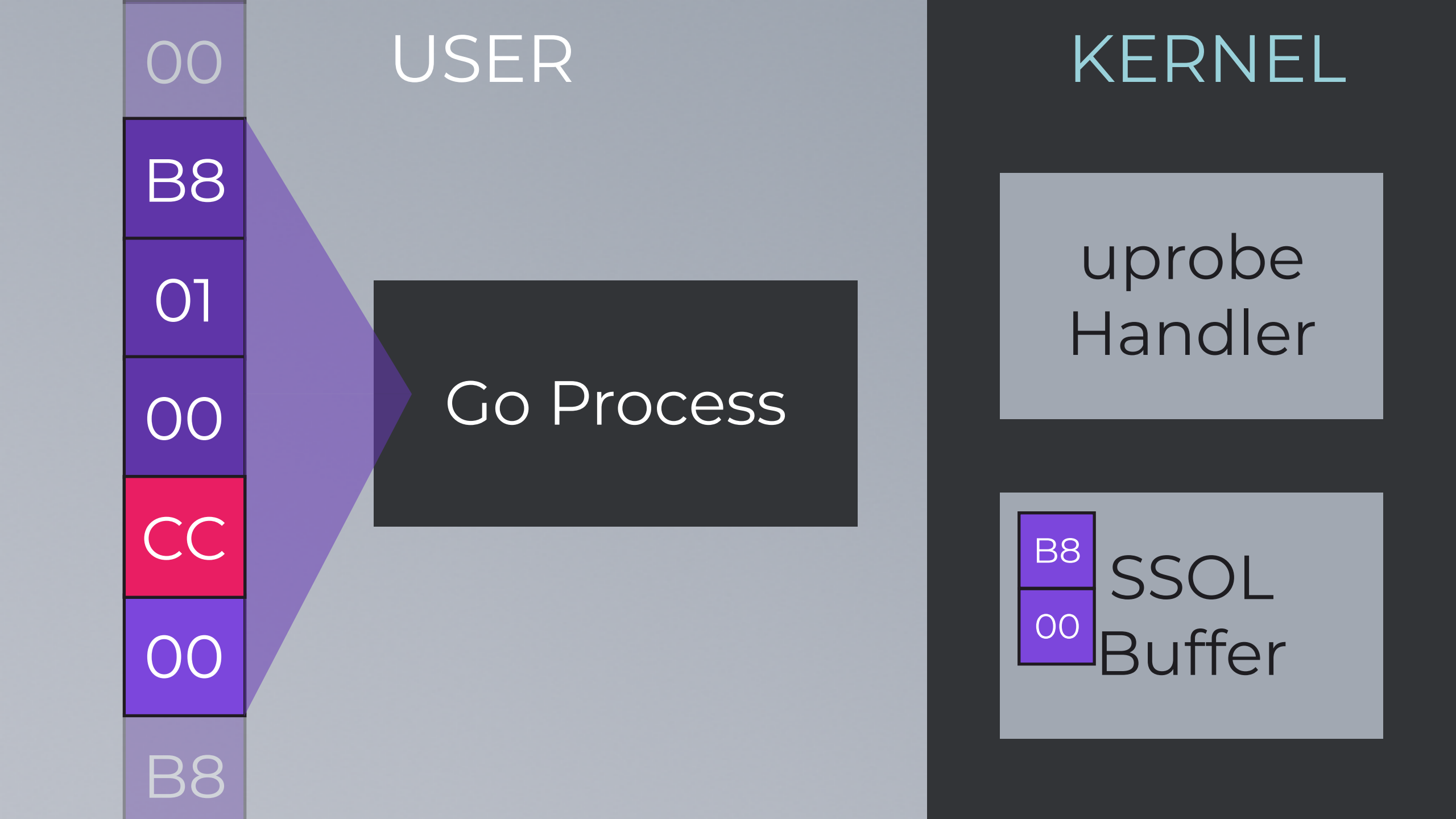
Go Process Running as Normal

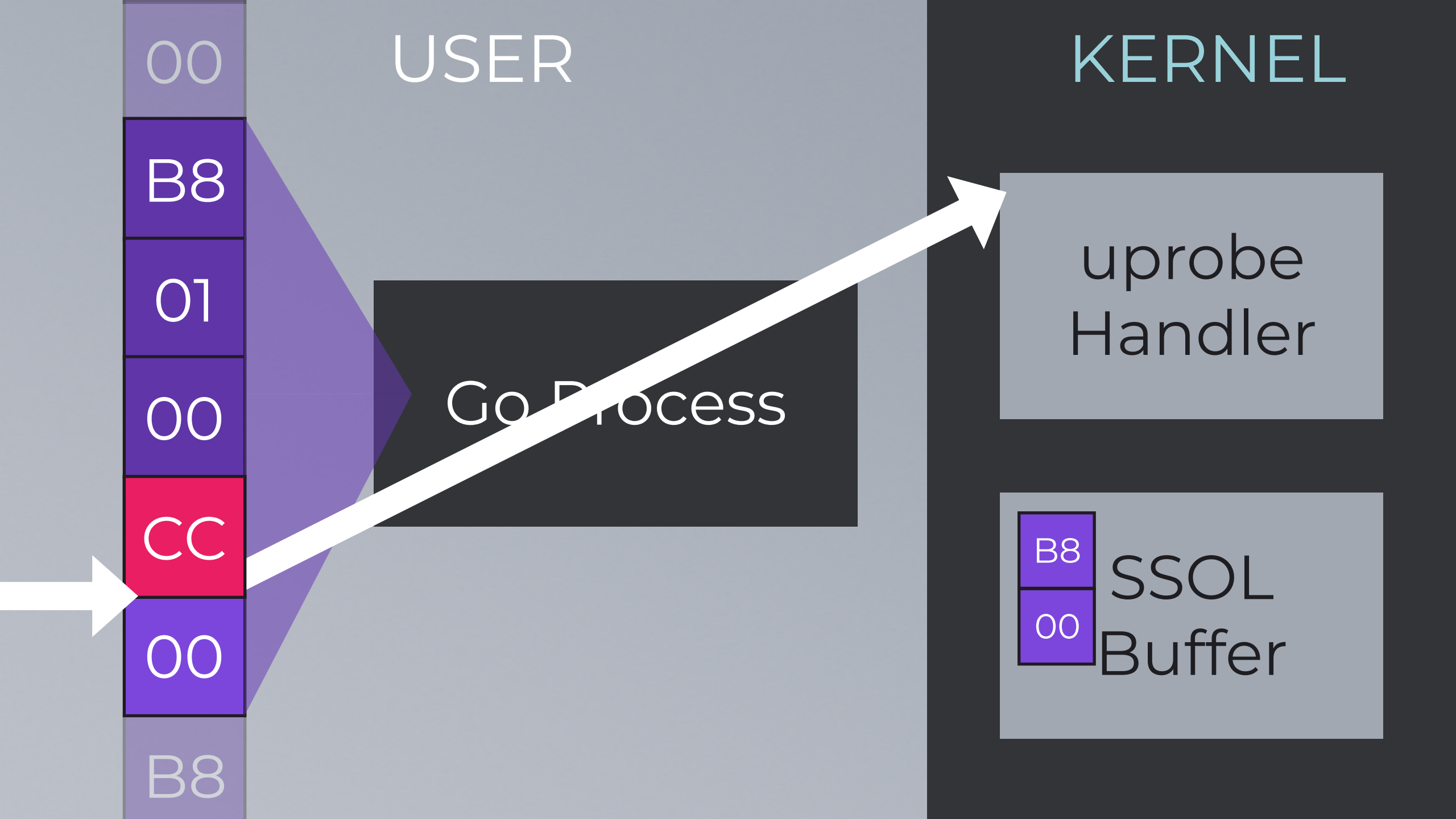

00
B8
01
00
B8
00
B8
USER
Go Process
KERNEL
uprobe
Handler
SSOL
Buffer

The Kernel Sets a Breakpoint

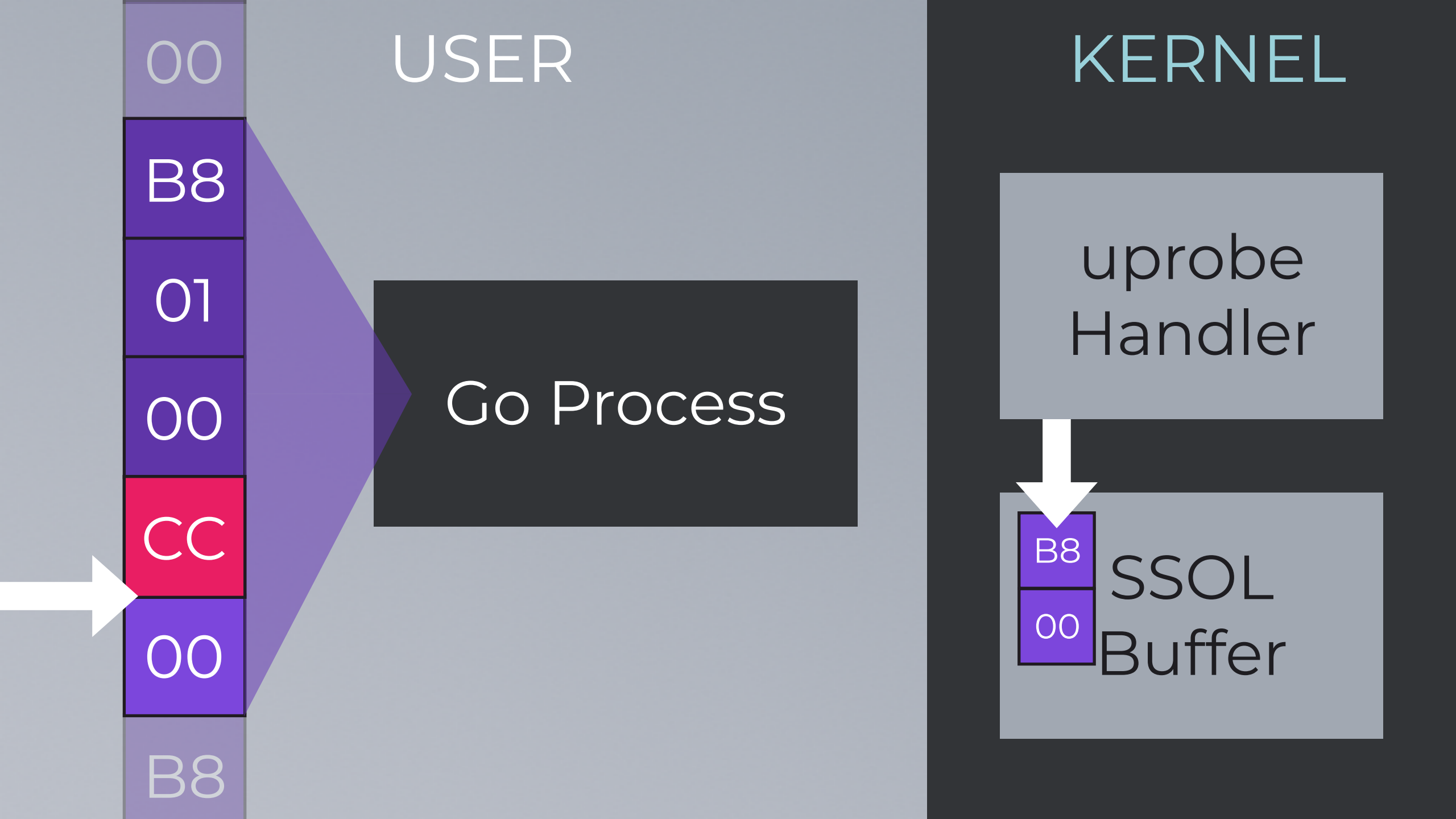

00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
uprobe
Handler
B8
00
SSOL
Buffer

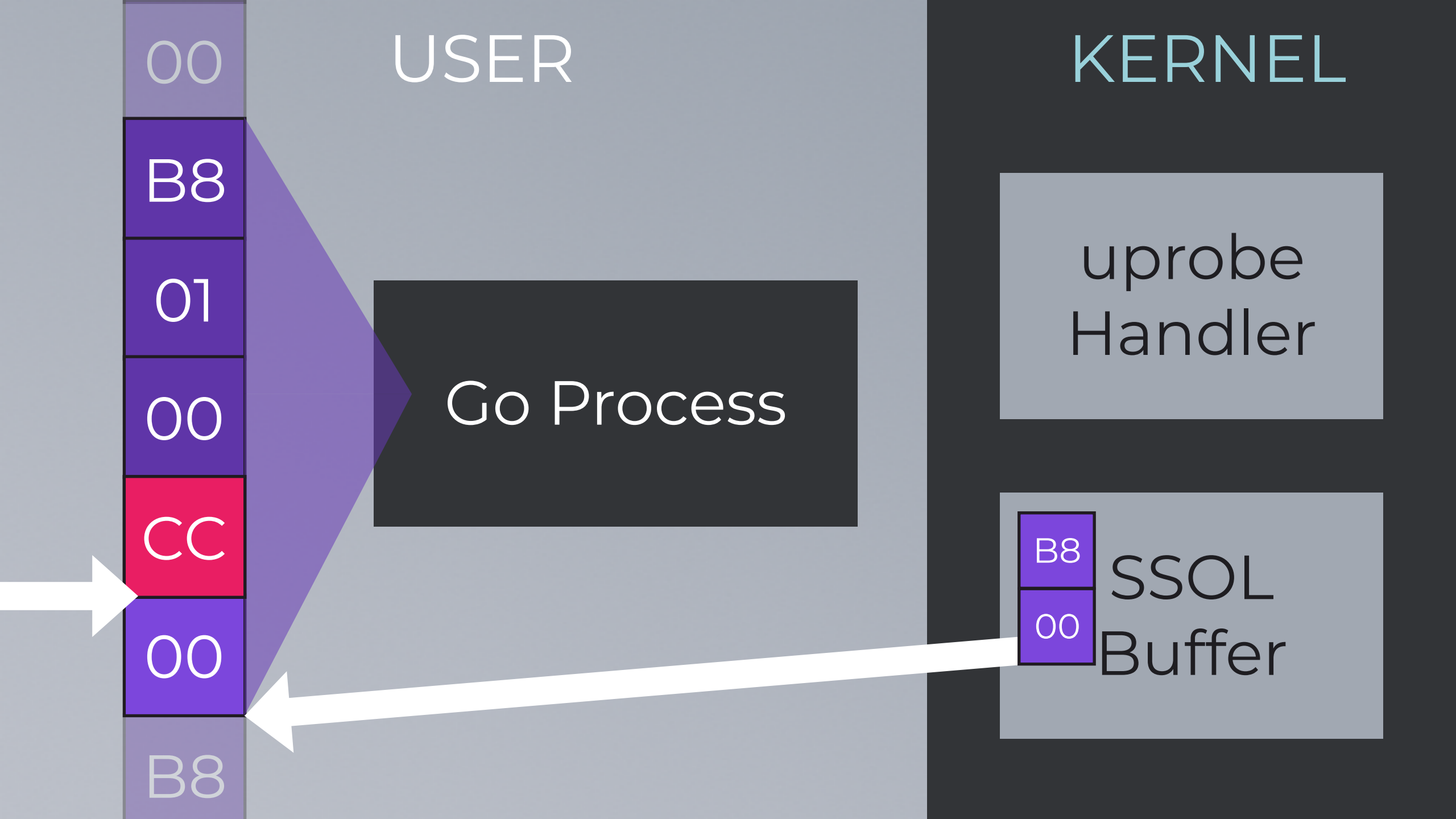

When This is Hit the
uprobe Handler Fires

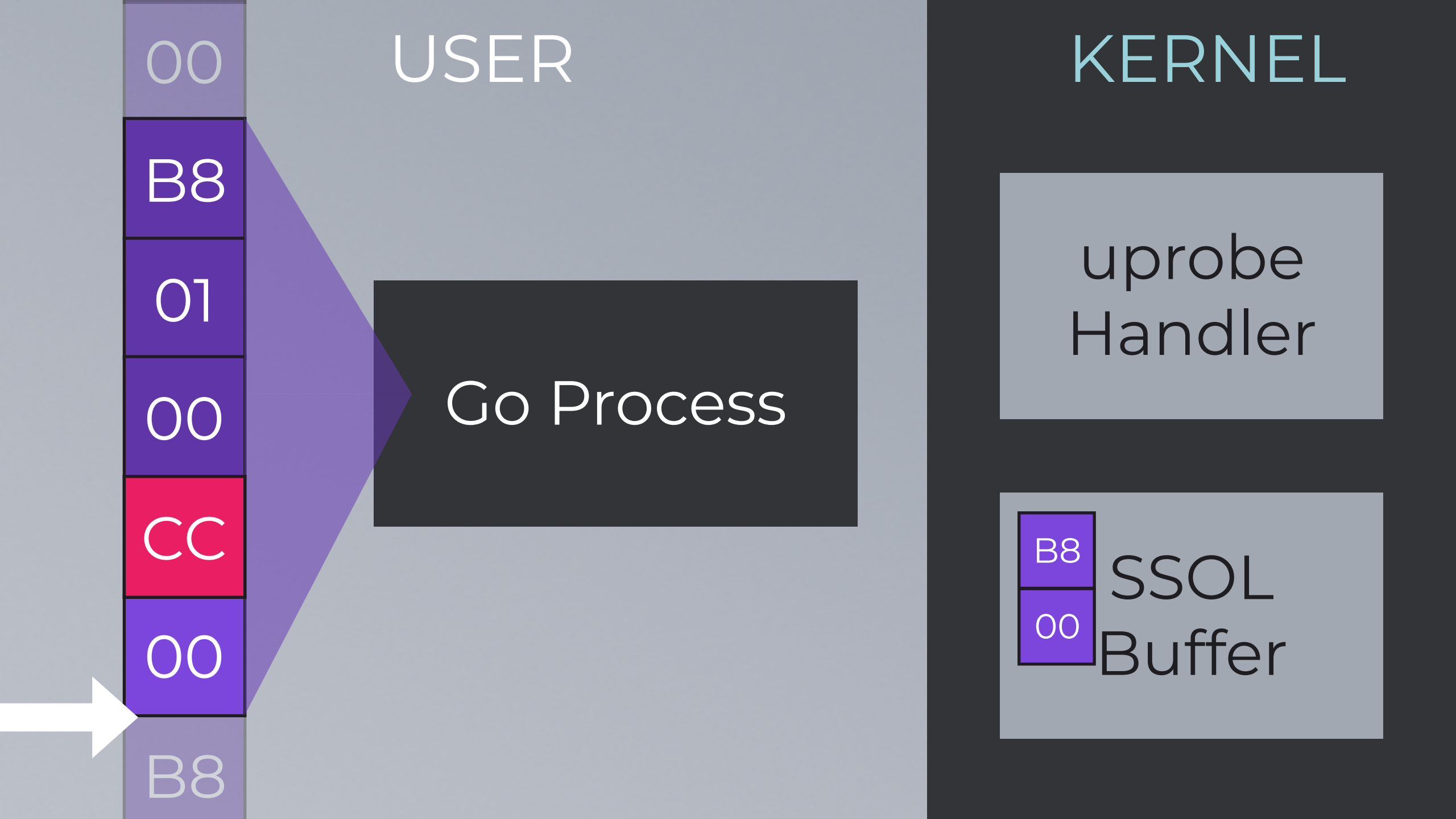

00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
uprobe
Handler
B8
00
SSOL
Buffer


00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
uprobe
Handler
B8
00
SSOL
Buffer

After the Handler Runs, The Copied Instruction is Single Stepped and the Program Resumes


00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
uprobe
Handler
B8
00
SSOL
Buffer


00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
uprobe
Handler
B8
00
SSOL
Buffer


00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
uprobe
Handler
B8
00
SSOL
Buffer

How do you write a uprobe handler?

Write a kernel module?
A custom kernel?

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, including uprobes.

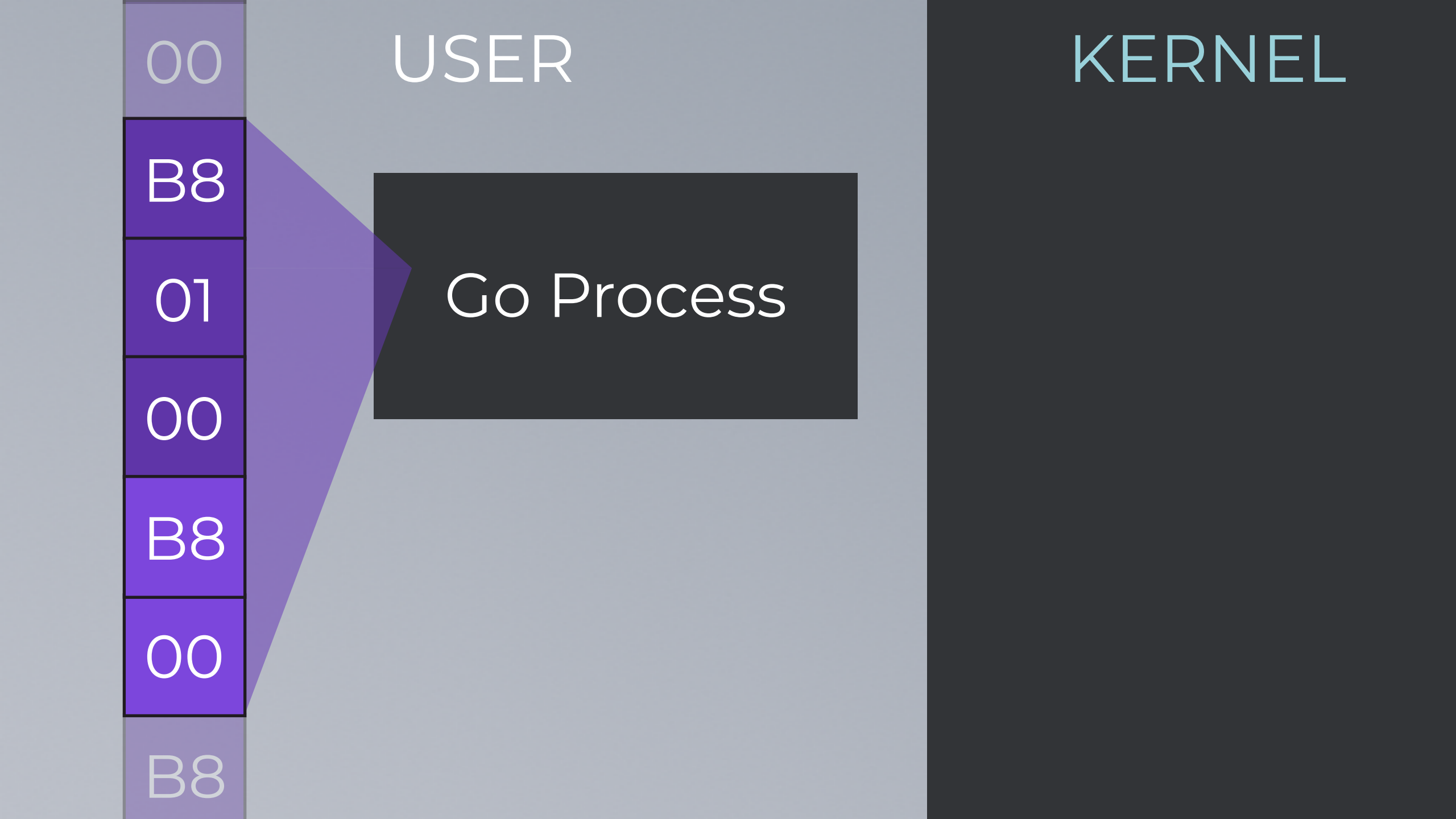
BCC

BCC is a compiler for BPF programs written in C

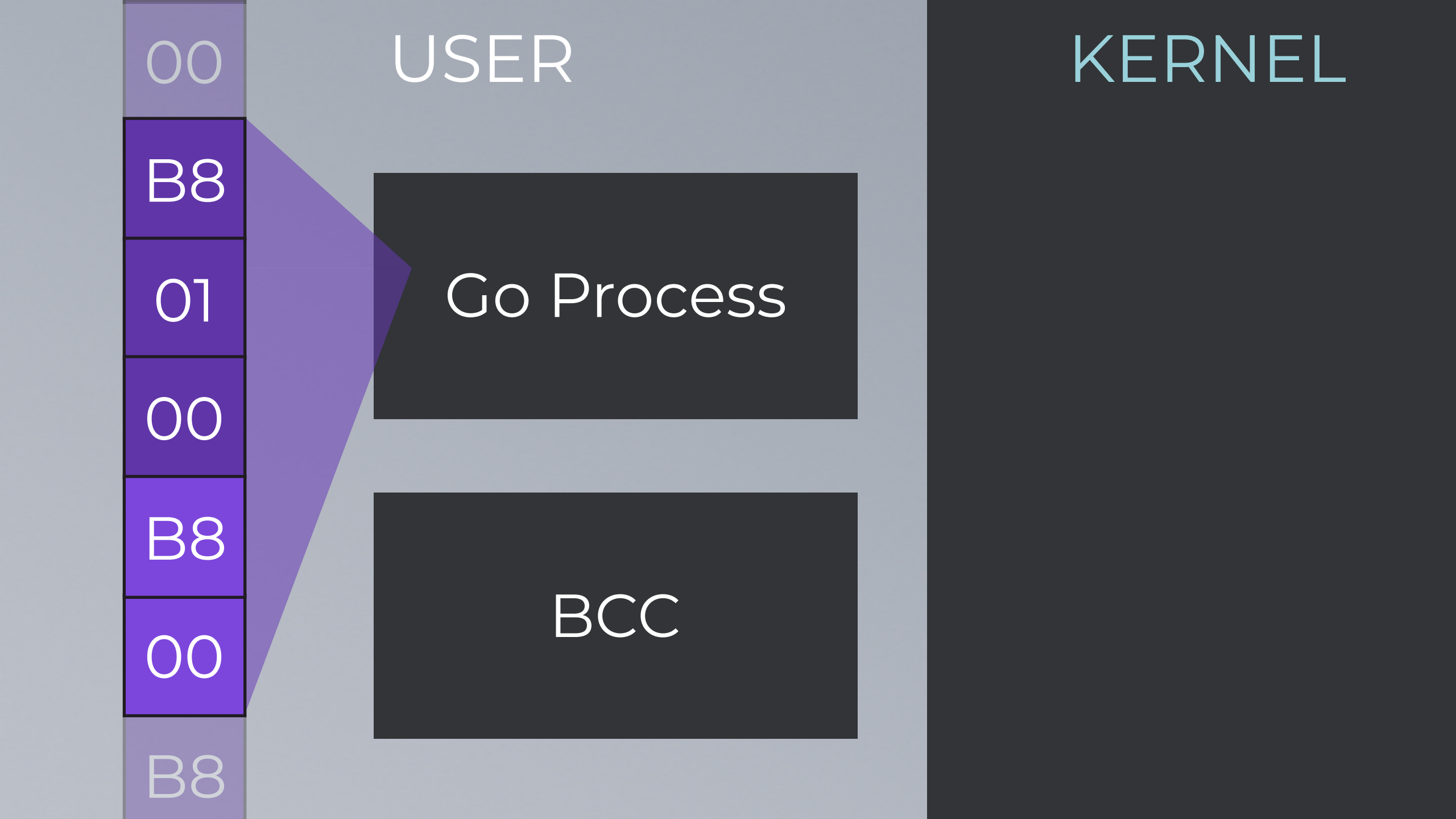
It assists with interacting with your BPF programs

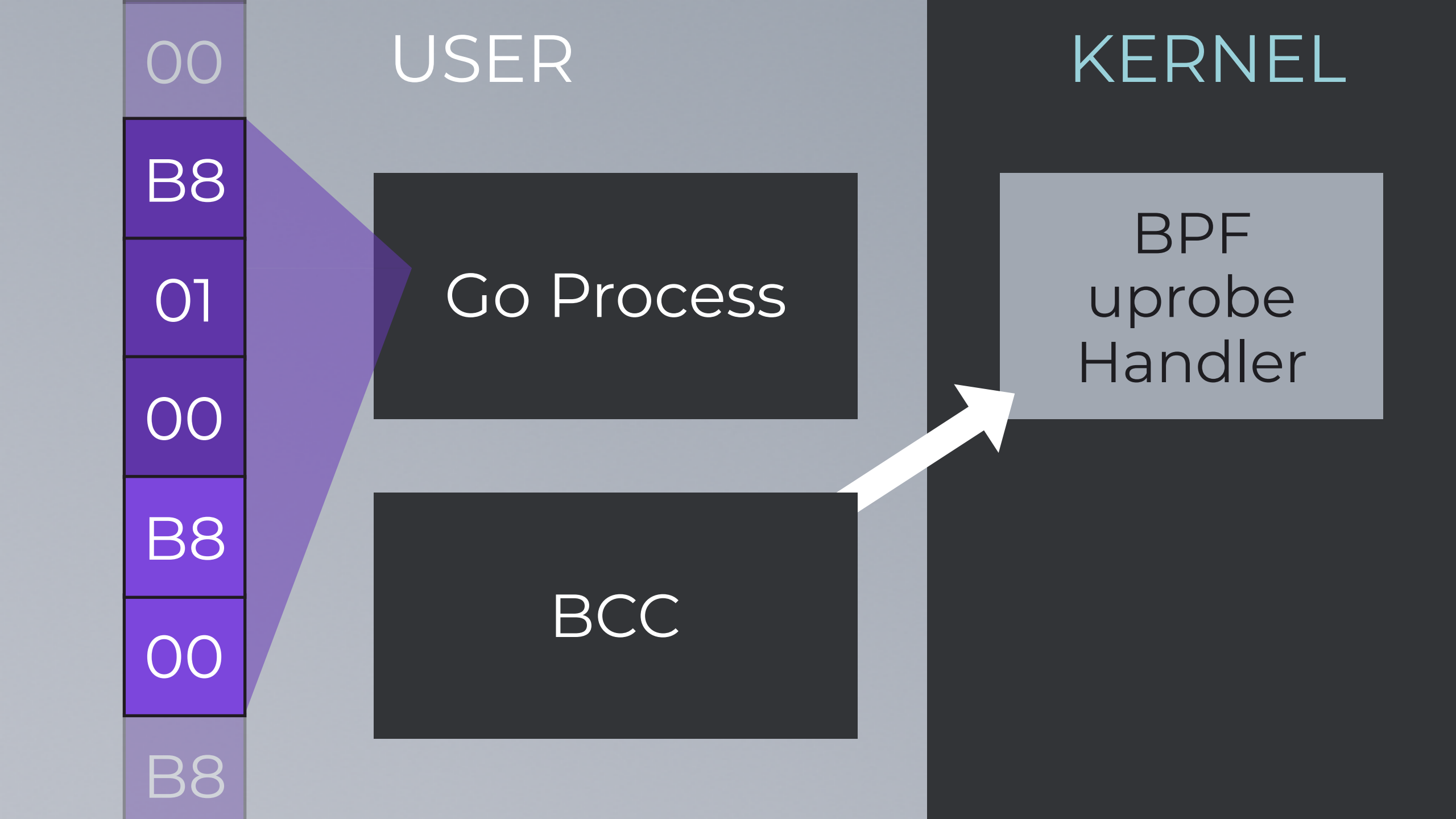
It is implemented as a library (libbcc.so)

Go Process Running as Normal

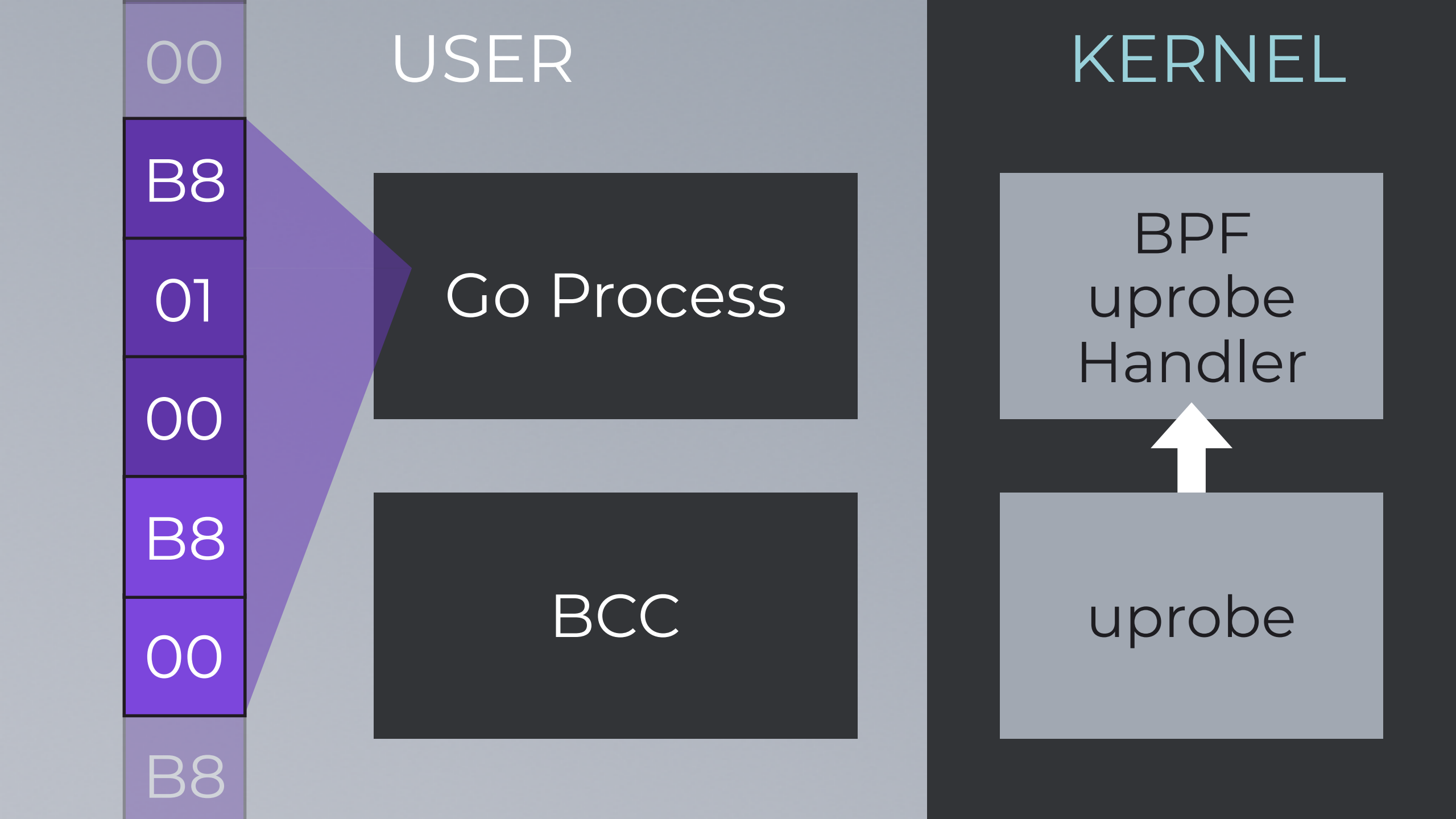

00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process

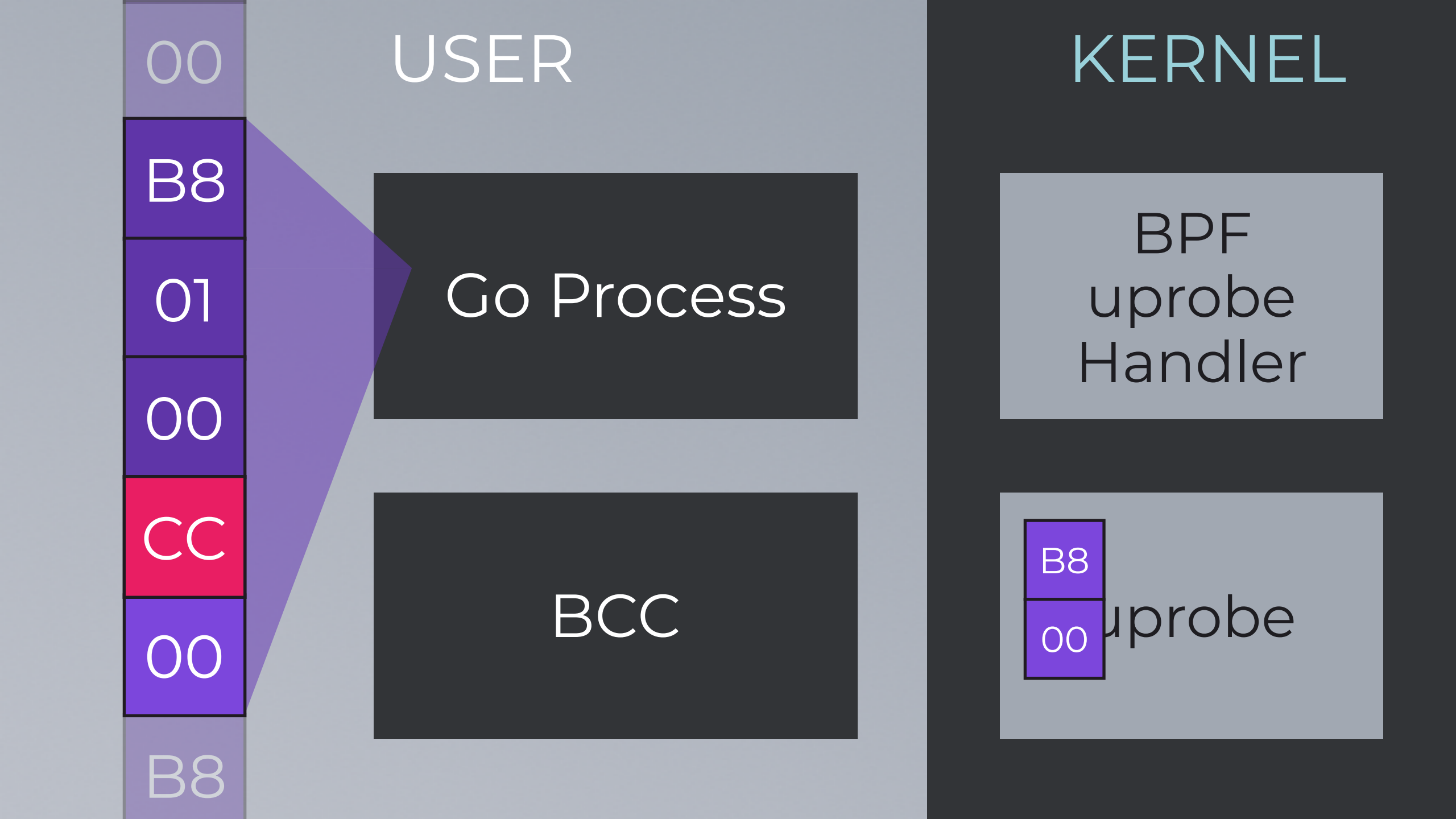
BCC Compiles and Injects BPF Program


00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
BCC

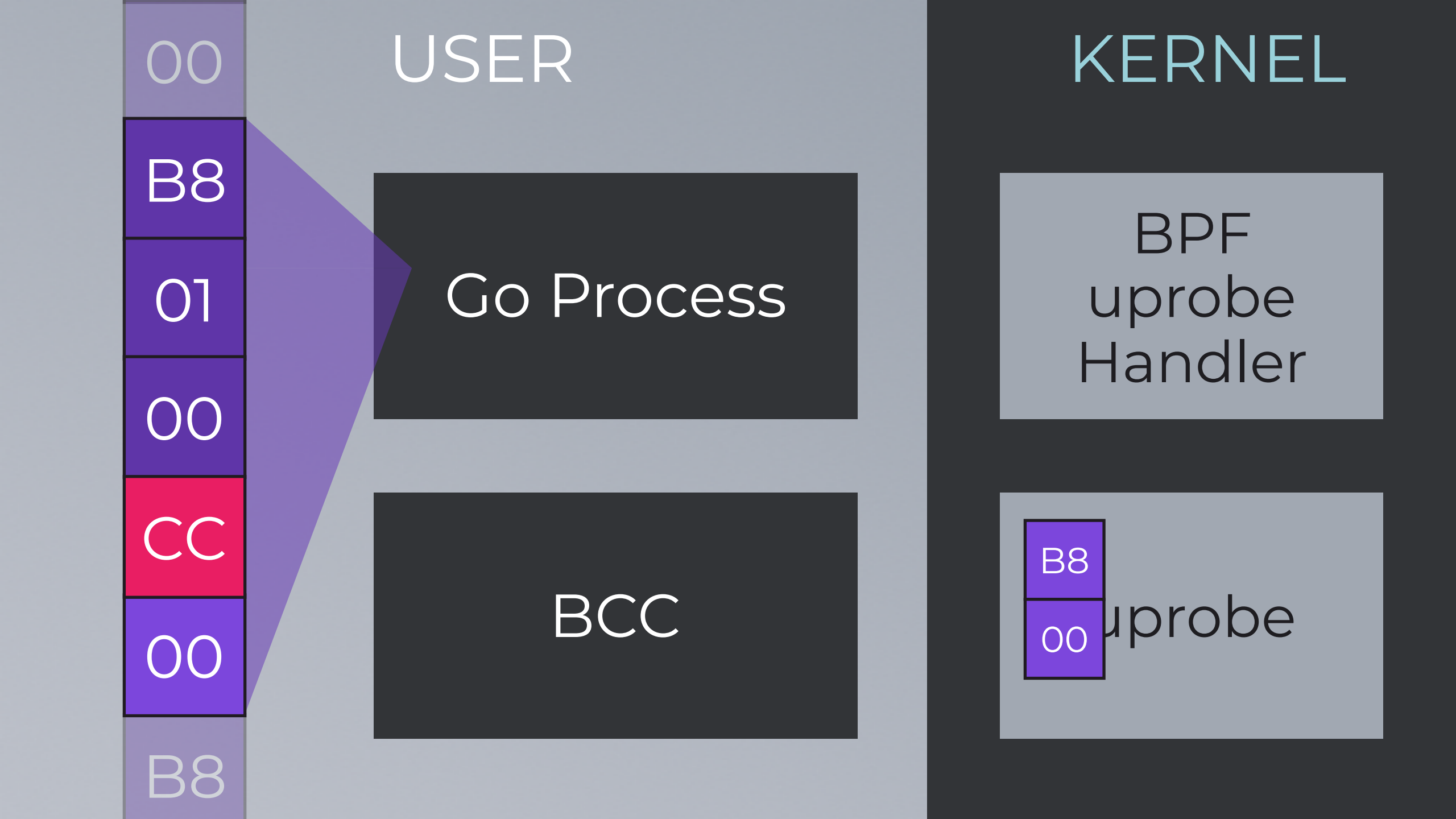

00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
BCC
BPF
uprobe
Handler

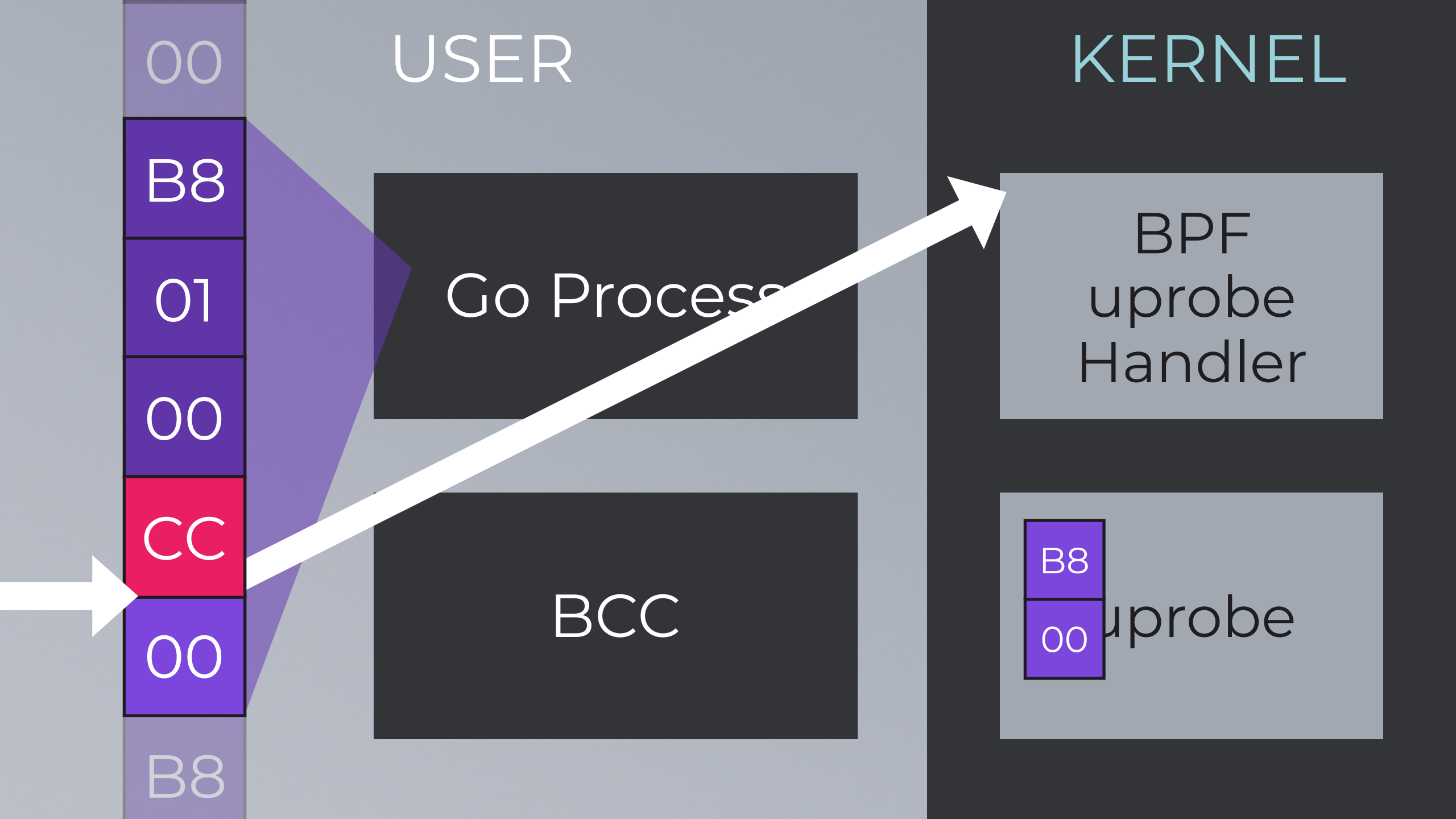
BCC Creates uprobe and Attaches it to the BPF Program

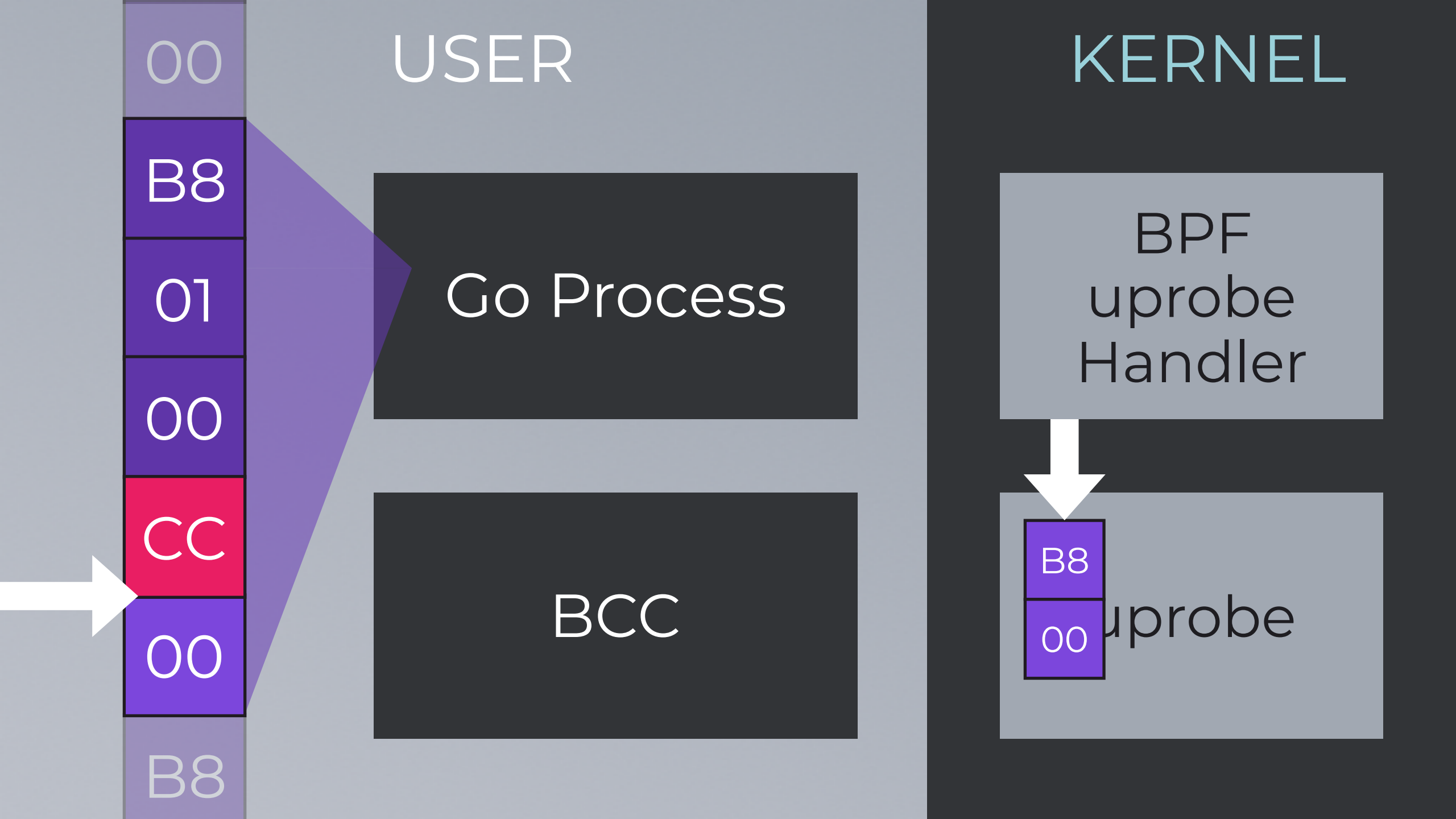

00
B8
01
00
B8
00
B8
USER
KERNEL
Go Process
BCC
BPF
uprobe
Handler
uprobe

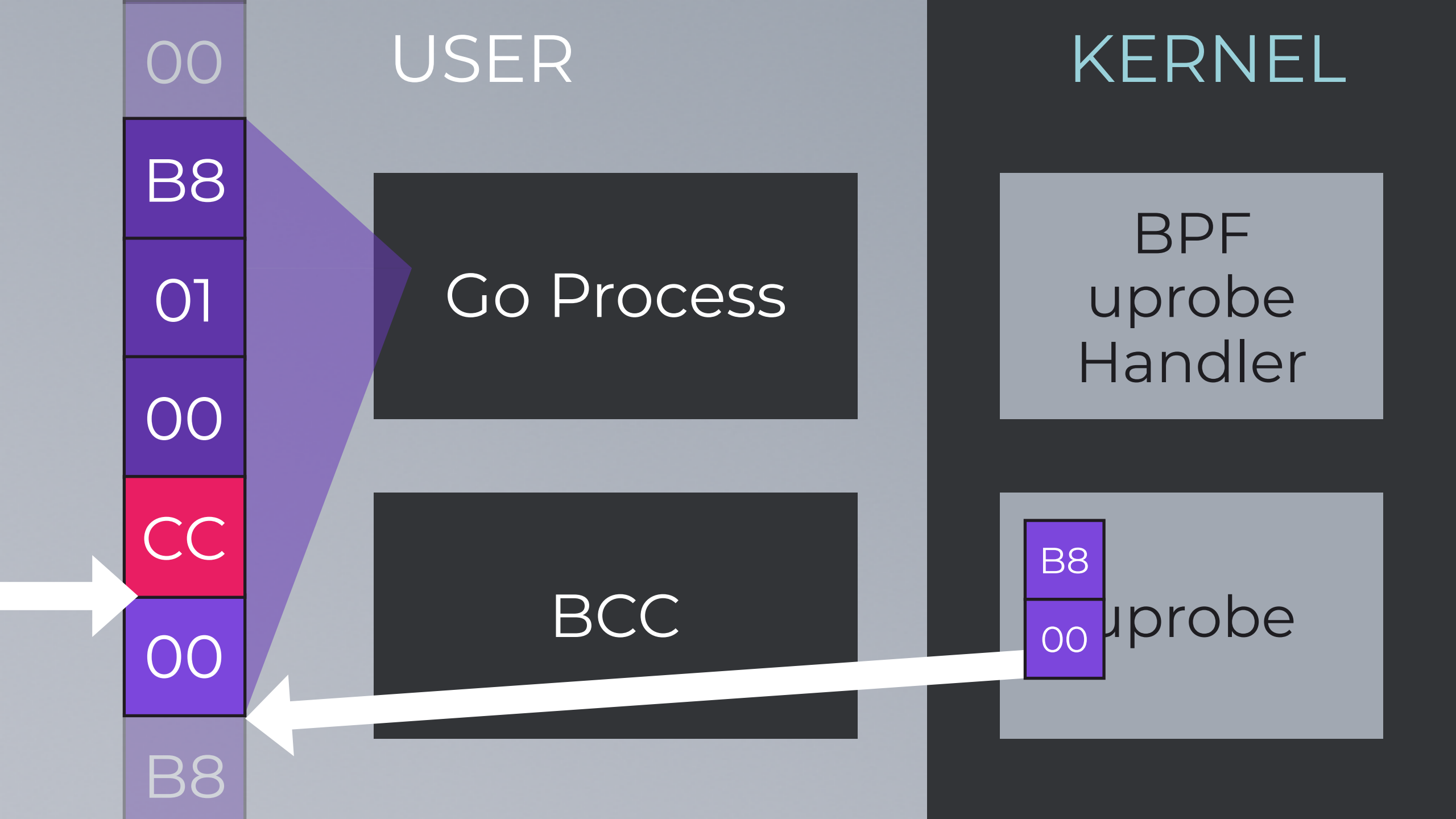

00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
BCC
BPF
uprobe
Handler
B8
00
uprobe

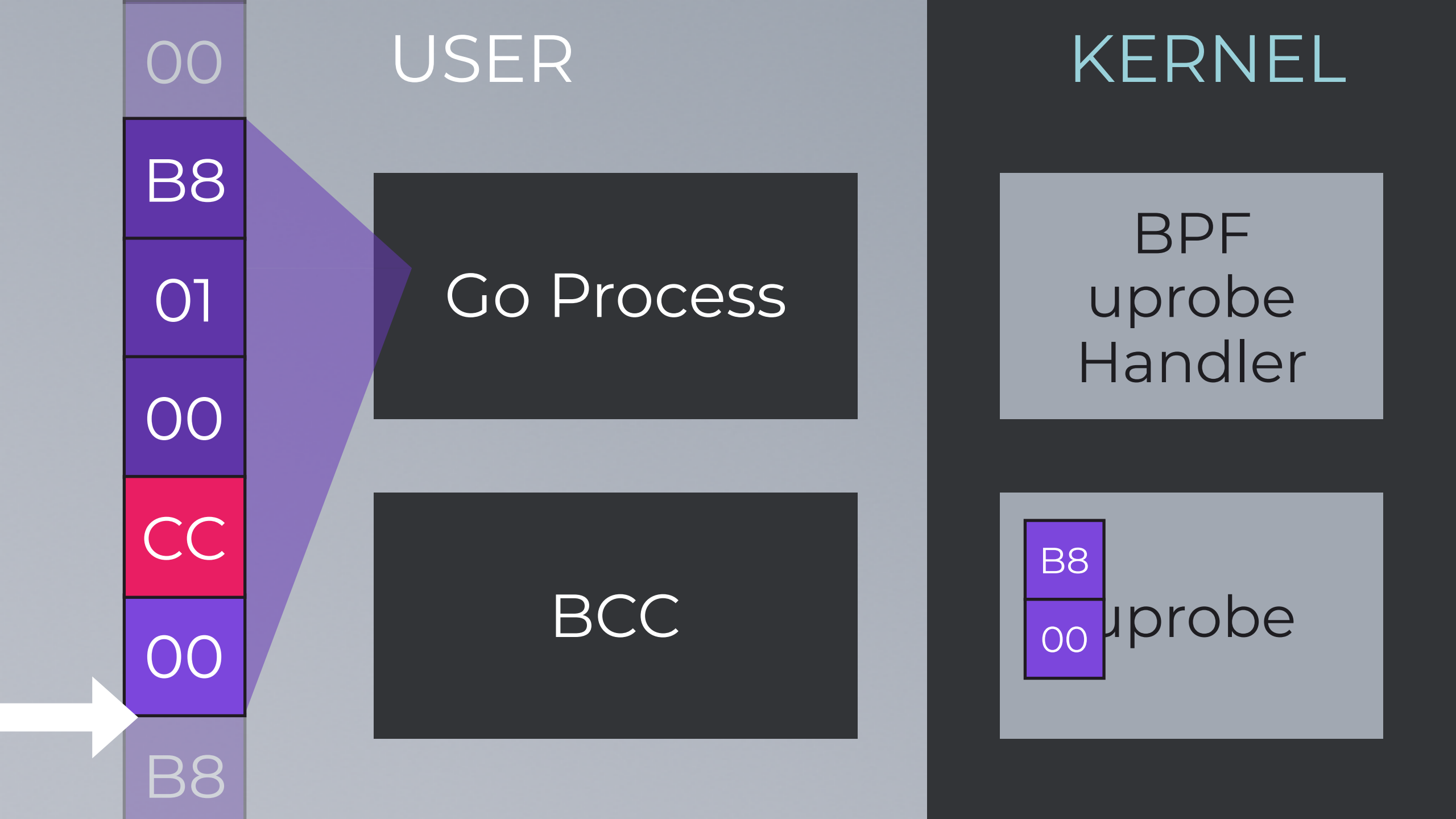
When uprobe is Hit the BPF Program is Ran


00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
BCC
BPF
uprobe
Handler
B8
00
uprobe

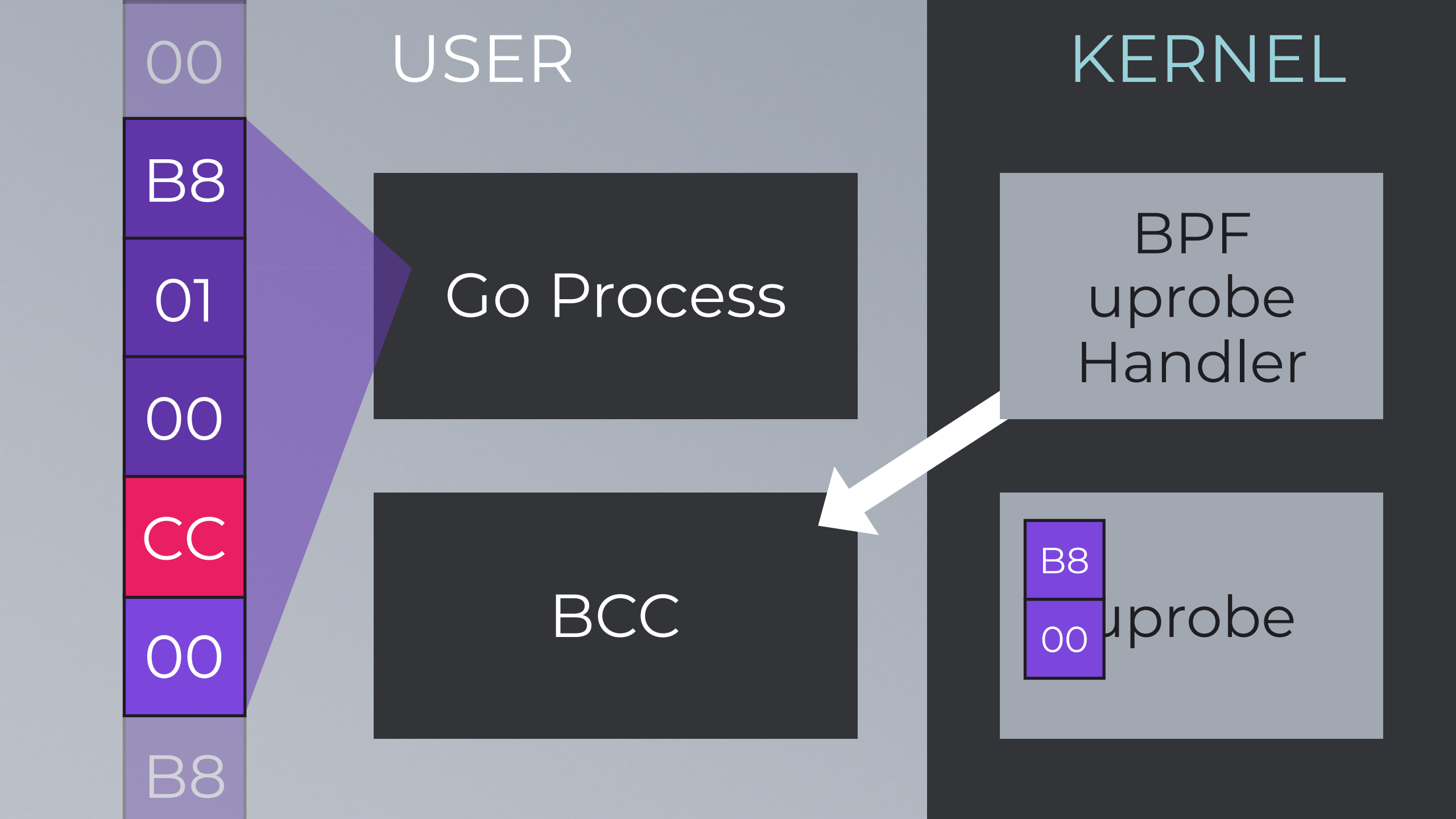

00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
BCC
BPF
uprobe
Handler
B8
00
uprobe


00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
BCC
BPF
uprobe
Handler
B8
00
uprobe


00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
BCC
BPF
uprobe
Handler
B8
00
uprobe


00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
BCC
BPF
uprobe
Handler
B8
00
uprobe

Asynchronously Data is Shared with the BCC Program


00
B8
01
00
CC
00
B8
USER
KERNEL
Go Process
BCC
BPF
uprobe
Handler
B8
00
uprobe

bpftrace

Let’s see an example of using bpftrace

```
uprobe:/tmp/counter:"main.doWork" {
    @ = *uaddr("main.counter");
}

interval:s:1 {
    printf("counter: %d\t %d ops/sec\n", @, @-@prev);
    @prev = @;
}
```

```
$ /tmp/counter --report
This is a blackbox. Read my counter.
counter: 32,350 (32,350 ops/s)
```

```
$ ./global.bt
```

It would be much better if we could write these programs in Go


HD
HISTORY.COM

goBPF

goBPF provides Go bindings for libbcc

You have to write your BPF programs in C

Let’s see an example of using goBPF

```
const bpfSource = `

// declare an array to share data between kernel and user space
BPF_ARRAY(count, u64, 1);

// define the function to handle the uprobe events
int read_counter()
{
    // get pointer to counter in program's memory
    u64 *counterPtr = (u64 *)%d;

    // get pointer to array in kernel memory
    int first = 0;
    u64 zero = 0, *val;
    val = count.lookup_or_init(&first, &zero);

    // copy value from program's memory to kernel memory
    bpf_probe_read(val, sizeof(*counterPtr), counterPtr);
}
`
```

```
const bpfSource = `

// declare an array to share data between kernel and user space
BPF_ARRAY(count, u64, 1);

// define the function to handle the uprobe events
int read_counter()
{
    // get pointer to counter in program's memory
    u64 *counterPtr = (u64 *)%d;

    // get pointer to array in kernel memory
    int first = 0;
    u64 zero = 0, *val;
    val = count.lookup_or_init(&first, &zero);

    // copy value from program's memory to kernel memory
    bpf_probe_read(val, sizeof(*counterPtr), counterPtr);
}
`
```

```
const bpfSource = `

// declare an array to share data between kernel and user space
BPF_ARRAY(count, u64, 1);

// define the function to handle the uprobe events
int read_counter()
{
    // get pointer to counter in program's memory
    u64 *counterPtr = (u64 *)%d;

    // get pointer to array in kernel memory
    int first = 0;
    u64 zero = 0, *val;
    val = count.lookup_or_init(&first, &zero);

    // copy value from program's memory to kernel memory
    bpf_probe_read(val, sizeof(*counterPtr), counterPtr);
}
`
```

```
const bpfSource = `

// declare an array to share data between kernel and user space
BPF_ARRAY(count, u64, 1);

// define the function to handle the uprobe events
int read_counter()
{
    // get pointer to counter in program's memory
    u64 *counterPtr = (u64 *)%d;

    // get pointer to array in kernel memory
    int first = 0;
    u64 zero = 0, *val;
    val = count.lookup_or_init(&first, &zero);

    // copy value from program's memory to kernel memory
    bpf_probe_read(val, sizeof(*counterPtr), counterPtr);
}
`
```

```
const bpfSource = `

// declare an array to share data between kernel and user space
BPF_ARRAY(count, u64, 1);

// define the function to handle the uprobe events
int read_counter()
{
    // get pointer to counter in program's memory
    u64 *counterPtr = (u64 *)%d;

    // get pointer to array in kernel memory
    int first = 0;
    u64 zero = 0, *val;
    val = count.lookup_or_init(&first, &zero);

    // copy value from program's memory to kernel memory
    bpf_probe_read(val, sizeof(*counterPtr), counterPtr);
}
`
```

```
const bpfSource = `

// declare an array to share data between kernel and user space
BPF_ARRAY(count, u64, 1);

// define the function to handle the uprobe events
int read_counter()
{
    // get pointer to counter in program's memory
    u64 *counterPtr = (u64 *)%d;

    // get pointer to array in kernel memory
    int first = 0;
    u64 zero = 0, *val;
    val = count.lookup_or_init(&first, &zero);

    // copy value from program's memory to kernel memory
    bpf_probe_read(val, sizeof(*counterPtr), counterPtr);
}
`
```

```
counterAddr := lookupSym("/tmp/counter", "main.counter")
```

```
func lookupSym(path, name string) uint64 {
   f, _ := elf.Open(path)
   syms, _ := f.Symbols()
   for _, s := range syms {
      if s.Name == name {
         return s.Value
      }
   }
   log.Fatalf("Unable to find sym: %s %s", path, name)
   return 0
}
```

```
m := bpf.NewModule(fmt.Sprintf(bpfSource, counterAddr), nil)
probe, _ := m.LoadUprobe("read_counter")
m.AttachUprobe("/tmp/counter", "main.doWork", probe, -1)
table := bpf.NewTable(m.TableId("count"), m)
```

```
var prev, count uint64
for {
    data, _ := table.Get([]byte{0})
    count = binary.LittleEndian.Uint64(data)
    fmt.Printf("counter: %d\t(%d ops/s)\n", count, count-prev)
    prev = count
    time.Sleep(time.Second)
}
```

```
$ /tmp/counter --report
```

```
$ /tmp/gobpf
```

We have three really great options:

delve bpftrace goBPF

	delve	bpftrace	goBPF
intercept any instruction	✓	✓	✓
understands go	✓	✗	✗
r/w memory	✓	✓	✓
r/w args/return vals	✓	✗	✓
read registers	✓	✓	✓
compute latencies	✓	✗	✗
can use go packages	✓	✗	✓
high frequency events	✗	✓	✓

Static instrumentation is great but does not provide total coverage of your programs.

Dynamic instrumentation allows you to observe every aspect of your program, as it runs, without it knowing.

There is no free lunch.

Delve is awesome and has a great API.

bpftrace and goBPF are excellent tools who's support for Go is continually improving.

With these new kinds of instrumentation we can achieve new levels of understanding about our programs!

I hope you find them as useful as I do!

Thank you!

wat.io

@jasonkeene

wat.io

@jasonkeene

wat.io/src

github.com/go-delve/delve

github.com/iovisor/bpftrace

github.com/iovisor/gobpf

github.com/iovisor/bcc