
It's the Kernel's Fault!

Custom Page Fault Handling With `bpf_fault`

Tal Zussman¹ Riju Dey¹ Hasan Zengin¹ Yiming Fang¹ David Hildenbrand^{2,3} Asaf Cidon¹

¹ Columbia University

² Technical University of Munich

³ Arm Germany GmbH



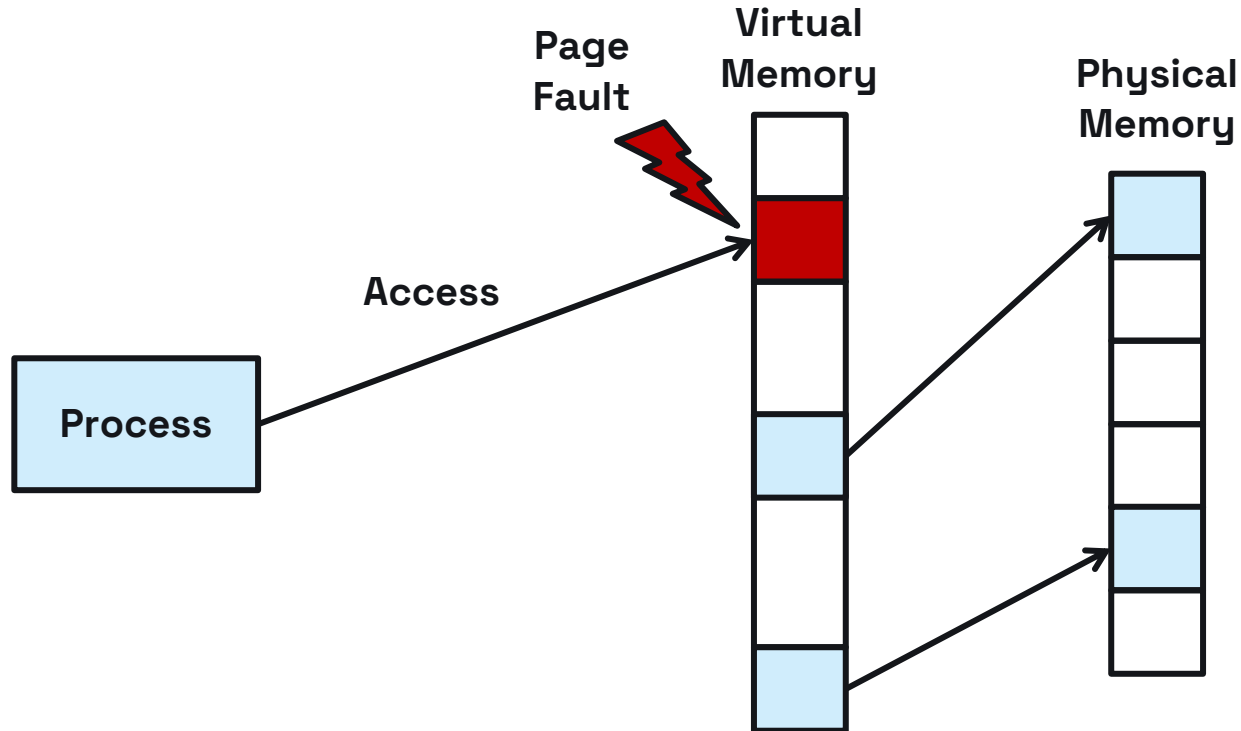
arm

TL;DR

- 01 Many applications want to handle their own page faults: virtual machines, garbage collectors, debuggers...
- 02 Linux's `userfaultfd` (uffd) handles faults in userspace, resulting in significant limitations and overhead
- 03 `bpf_fault` increases application performance and enables new use cases due to a streamlined, eBPF-based design

When do page faults happen?

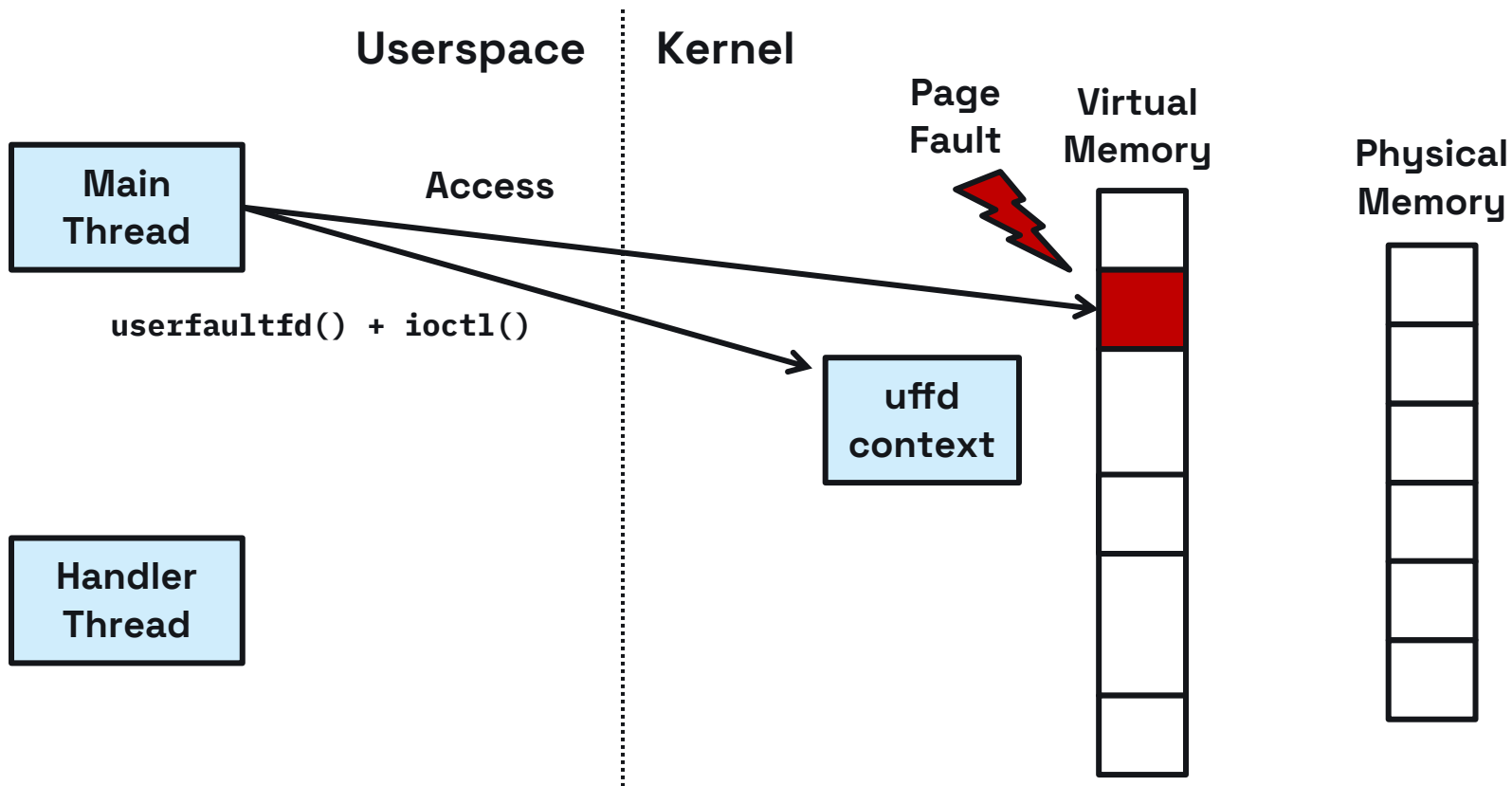
Ex: No page table entry (PTE) installed



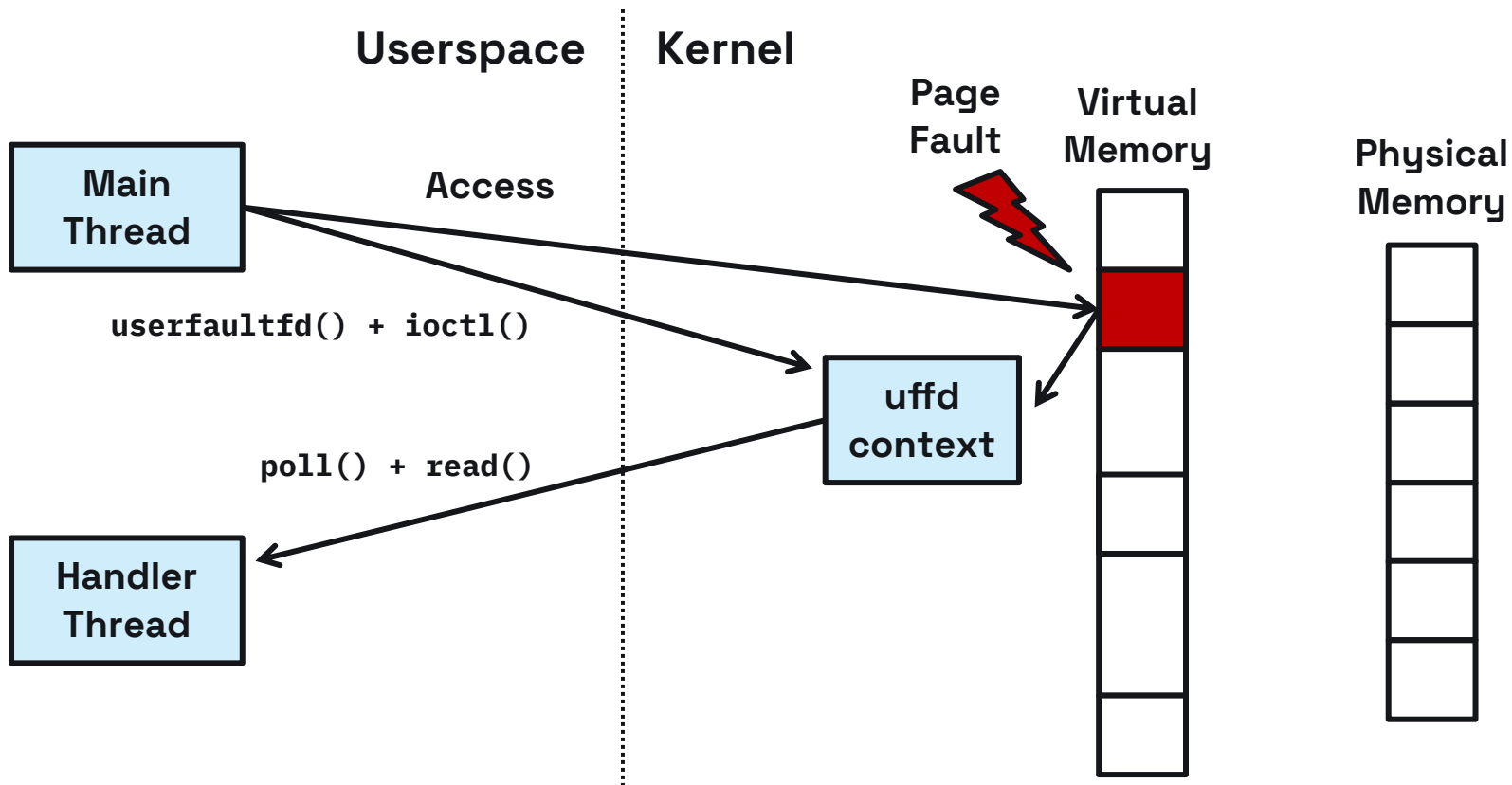
Why do we need custom fault handling?

- Set page contents on fault-in
 - VM migration, VM snapshot restore, memory tiering...
- Page access notifications
 - VM live snapshots, memory debugging, garbage collection...
- **Greater application control of memory management**

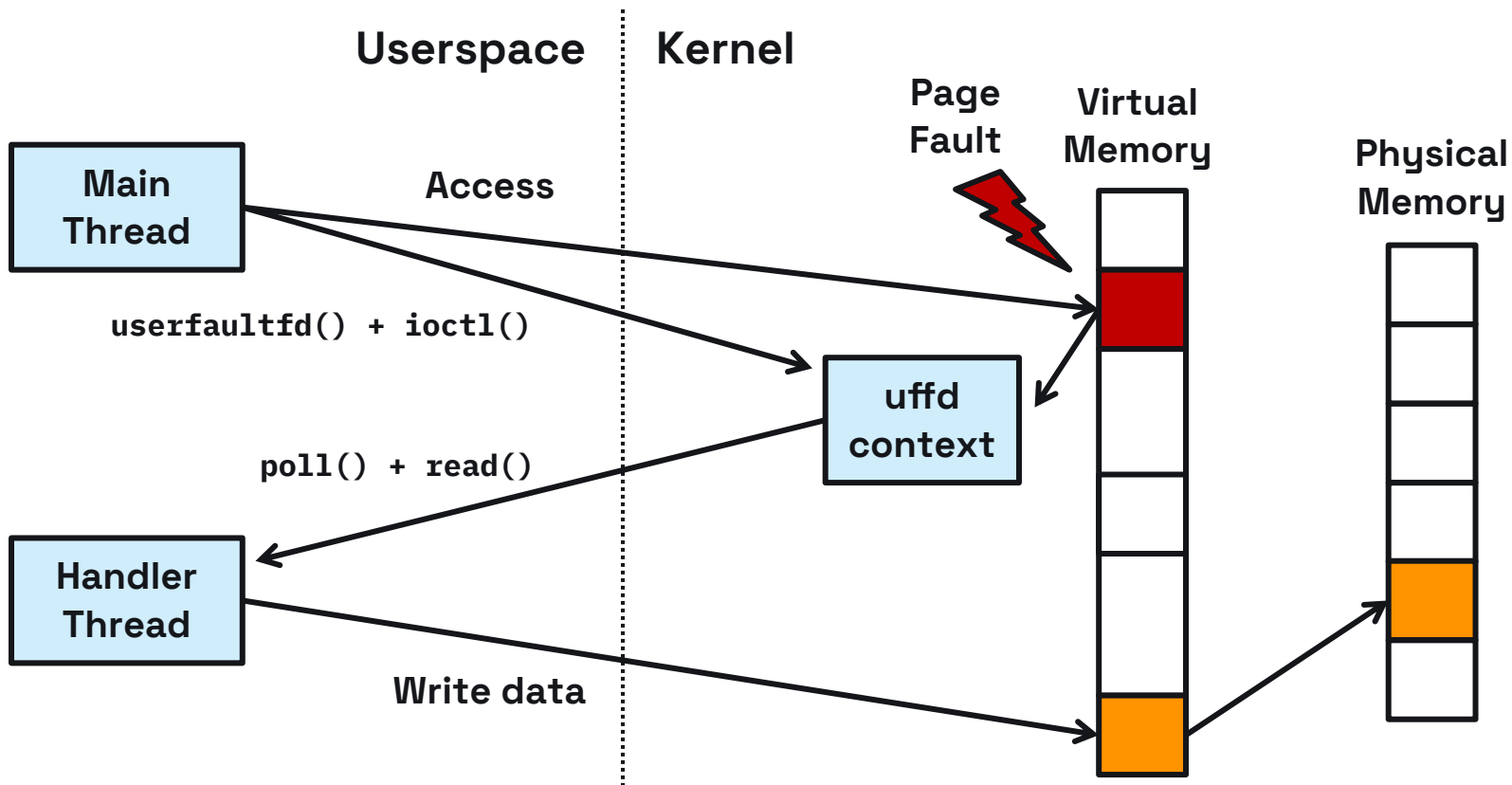
userfaultfd: offload page faults to userspace



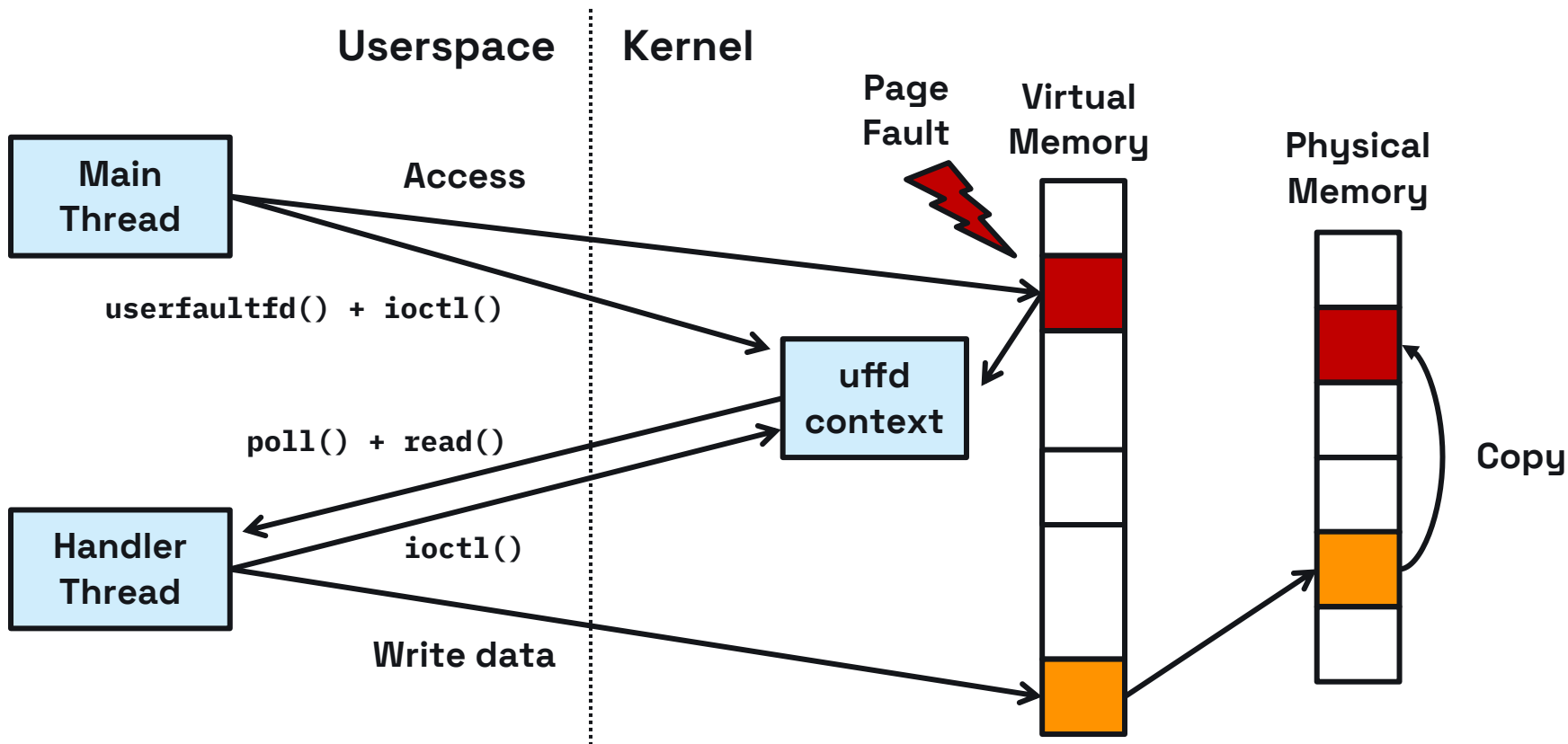
userfaultfd: offload page faults to userspace



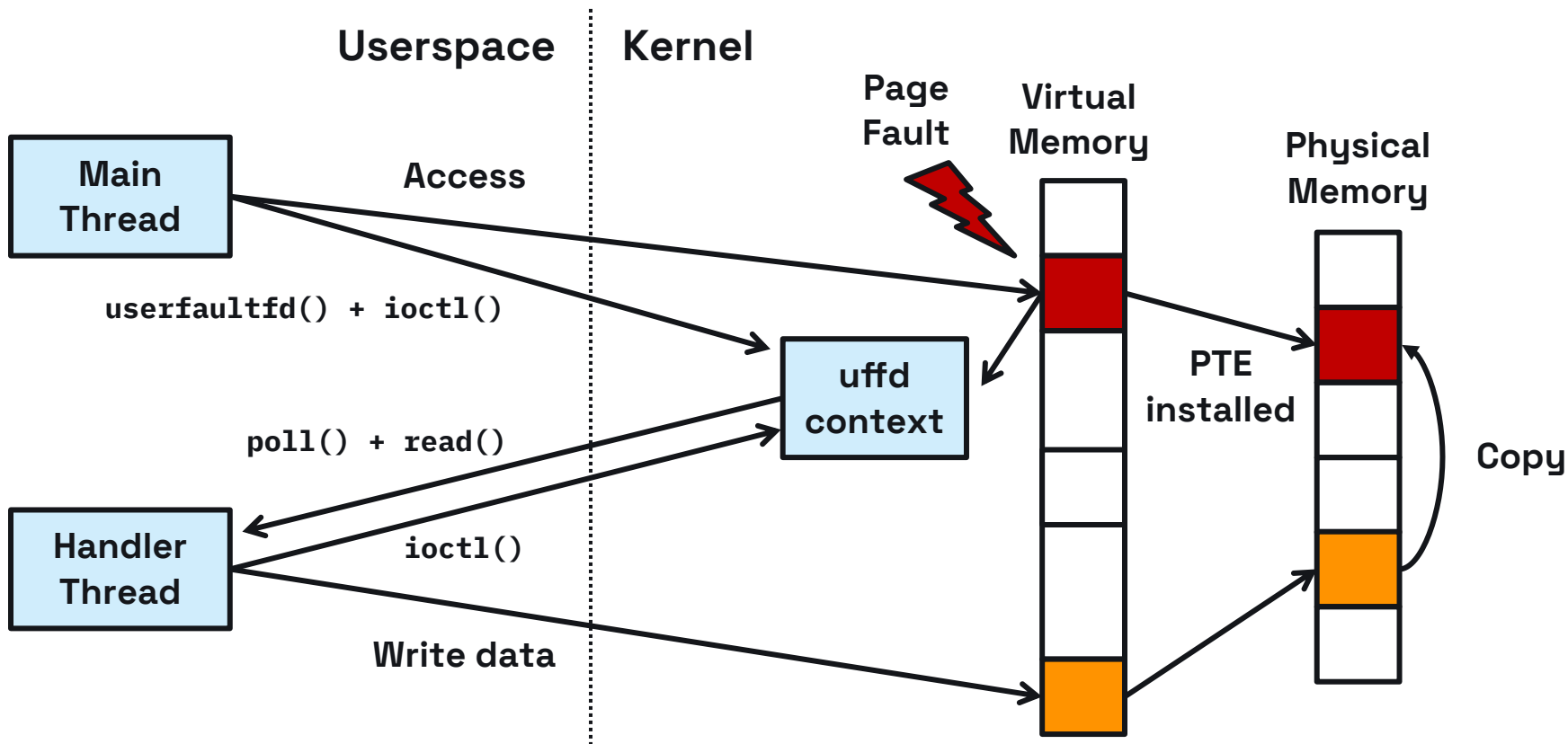
userfaultfd: offload page faults to userspace



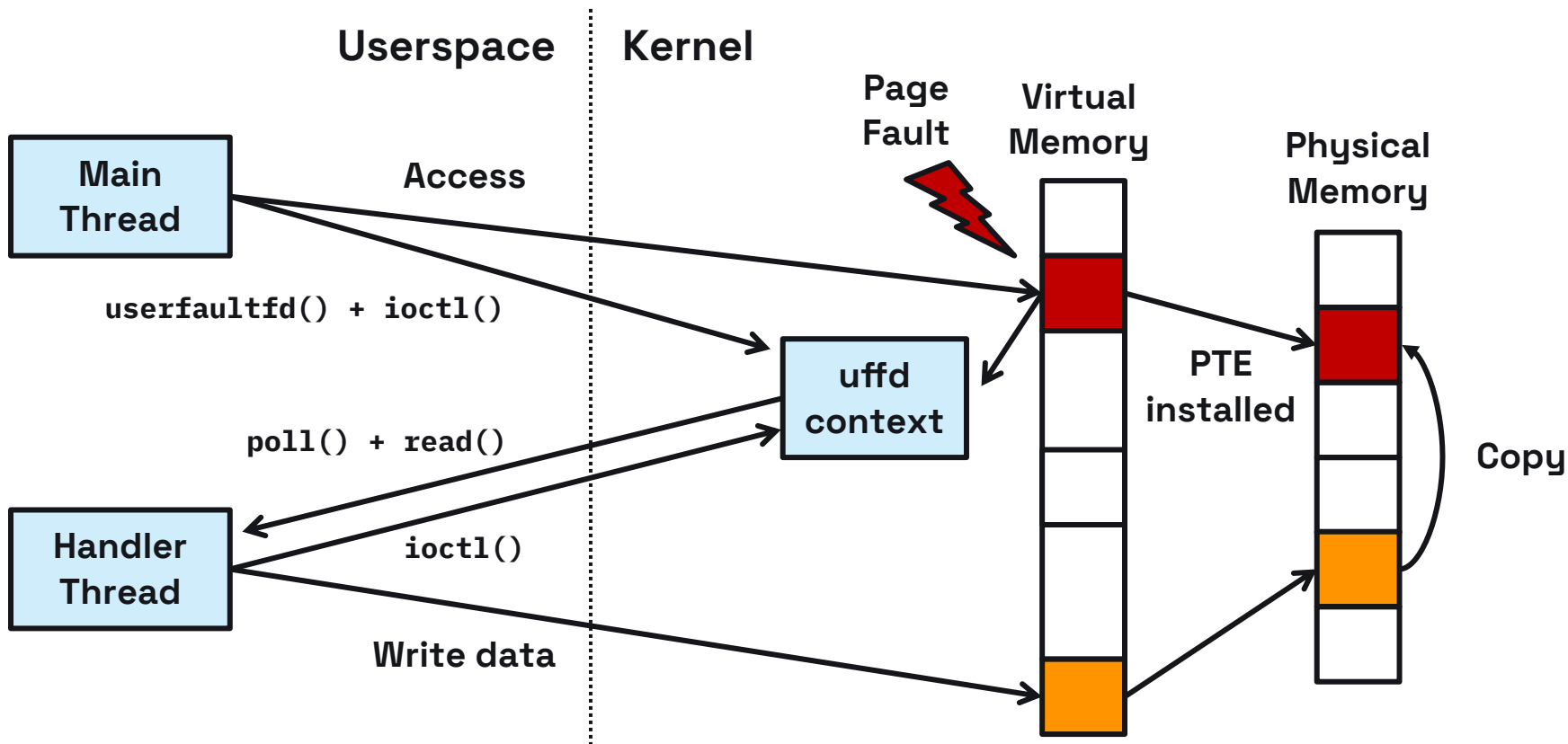
userfaultfd: offload page faults to userspace



userfaultfd: offload page faults to userspace



userfaultfd: offload page faults to userspace



Lots of back and forth...

userfaultfd has some downsides

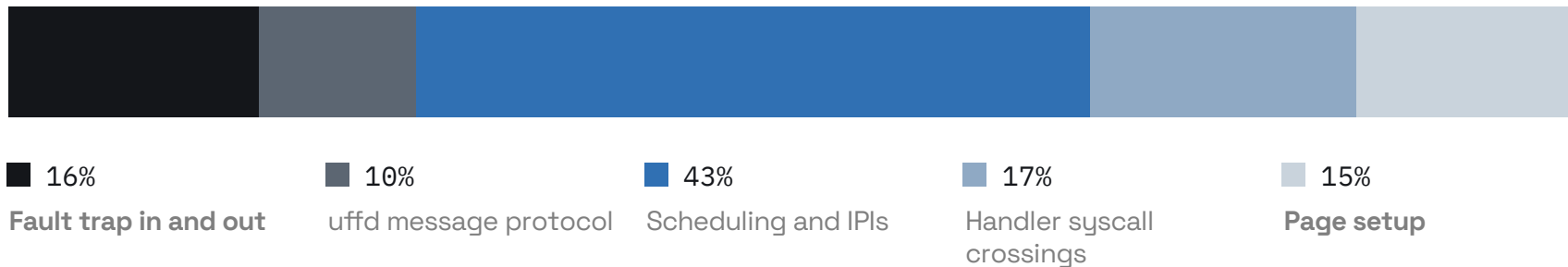
01 Overhead

02 Scalability

03 Applicability

04 Security

Most of a userfaultfd fault is overhead

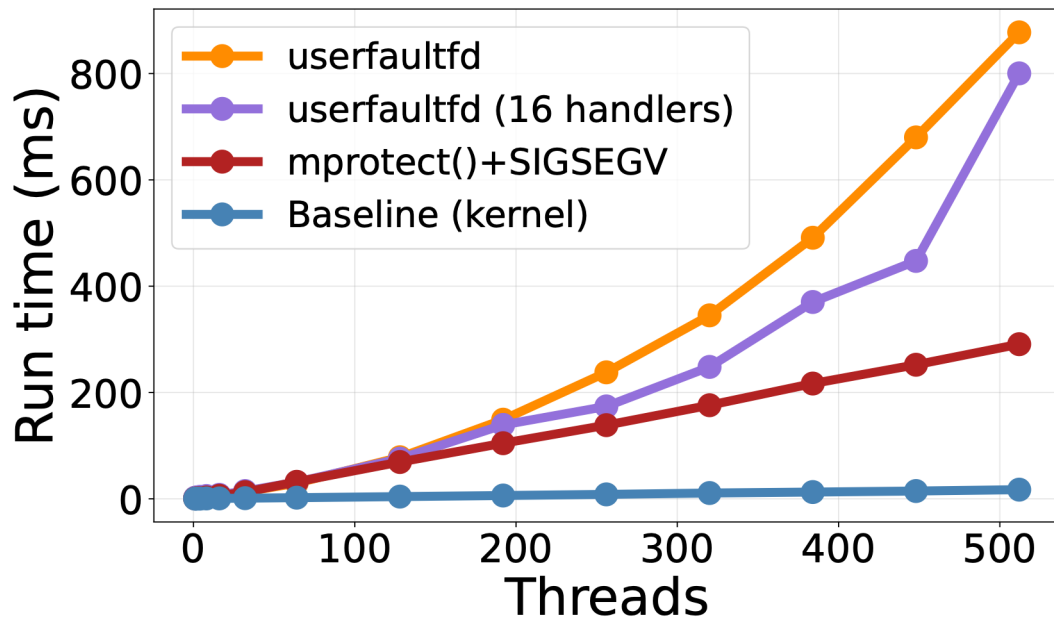


13.6× slower

than a plain kernel fault
(10.2 μ s vs. 0.75 μ s)

**Lots of kernel crossings, data copying,
context switches...**

Poor scalability due to the handler thread



616× increase

from 1 to 512 threads, compared to
57× for the baseline

Fault-handling thread limits applicability

- We don't necessarily want to have a thread spawned by a uffd-using library for the duration of the process
 - Want to be transparent to the process
 - May break on `fork()`
 - Non-cooperative mode exists, but turns into bottleneck

userfaultfd has some downsides

01 Overhead

02 Scalability

03 Applicability

04 Security



In the paper

**What if we could just remove
all the complexity?**

Let's push fault handling back into the kernel

- Need to run custom code in the kernel
- Attached to a virtual memory area (VMA)
- Different behavior for different fault types (missing, minor, WP)

bpf_fault: Let's push fault handling back into the kernel

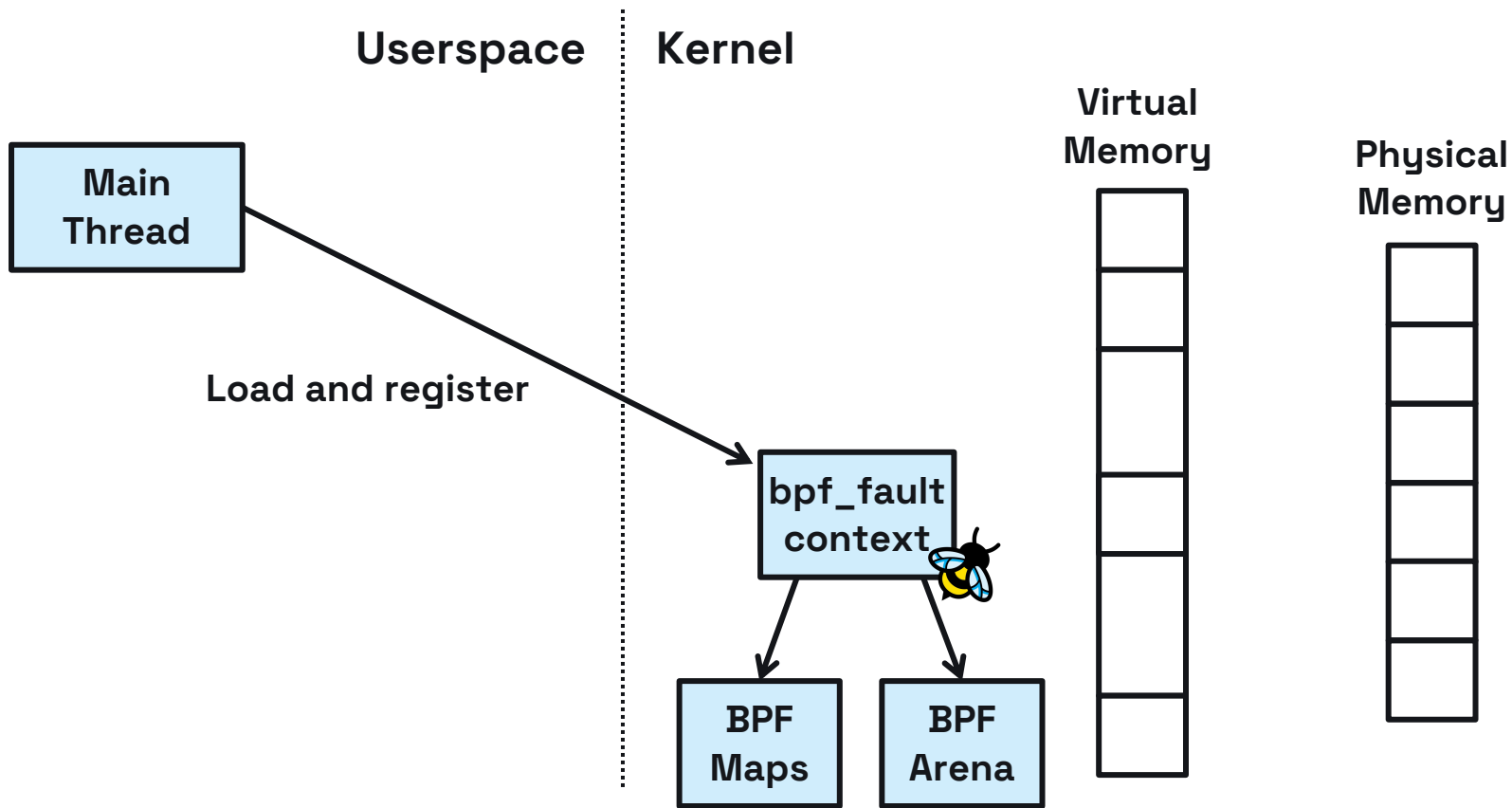
- Need to run custom code in the kernel
- Attached to a virtual memory area (VMA)
- Different behavior for different fault types (missing, minor, WP)



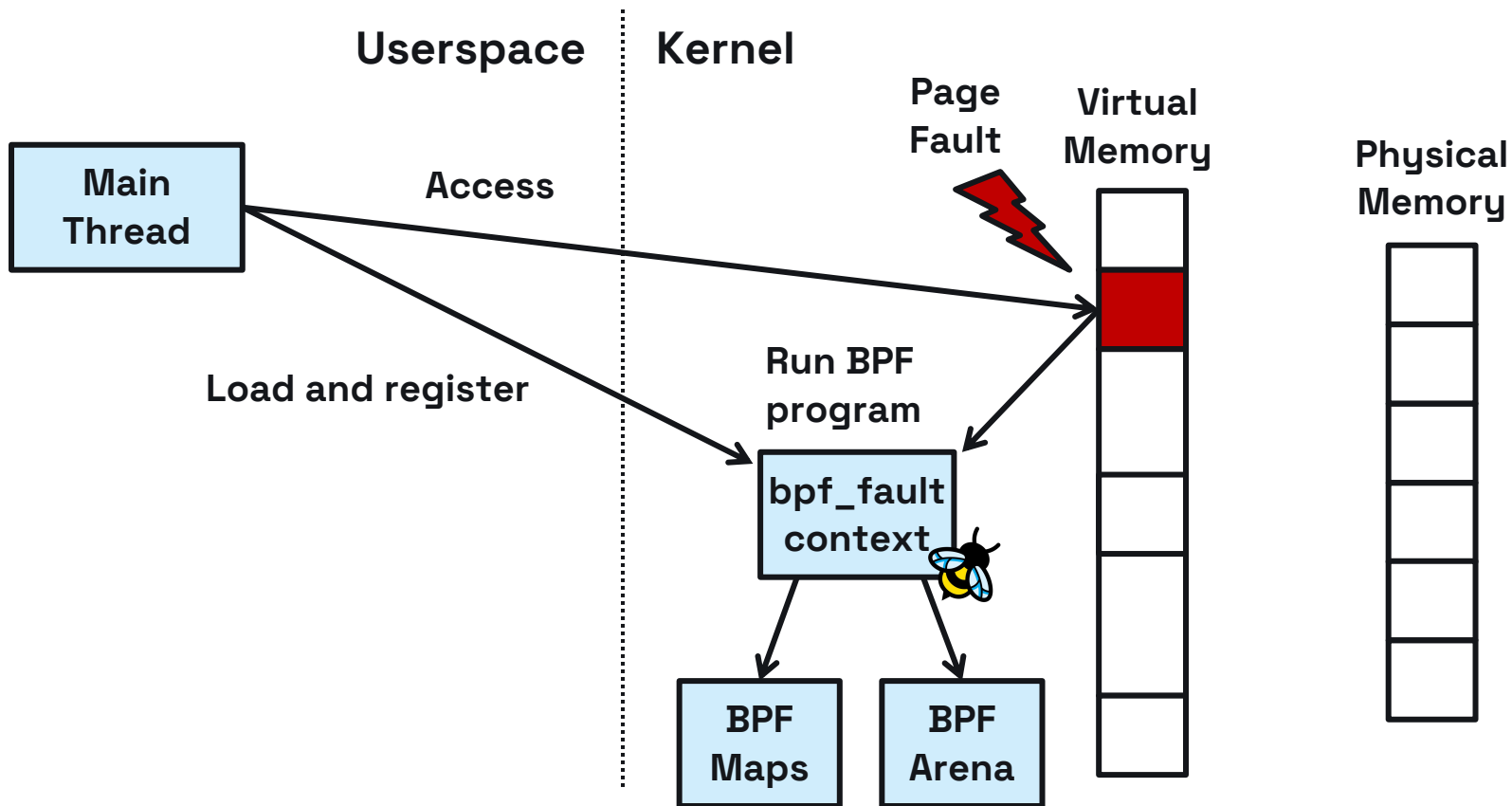
eBPF!

- Safely run userspace code in the kernel
- eBPF hooks in the fault handling paths

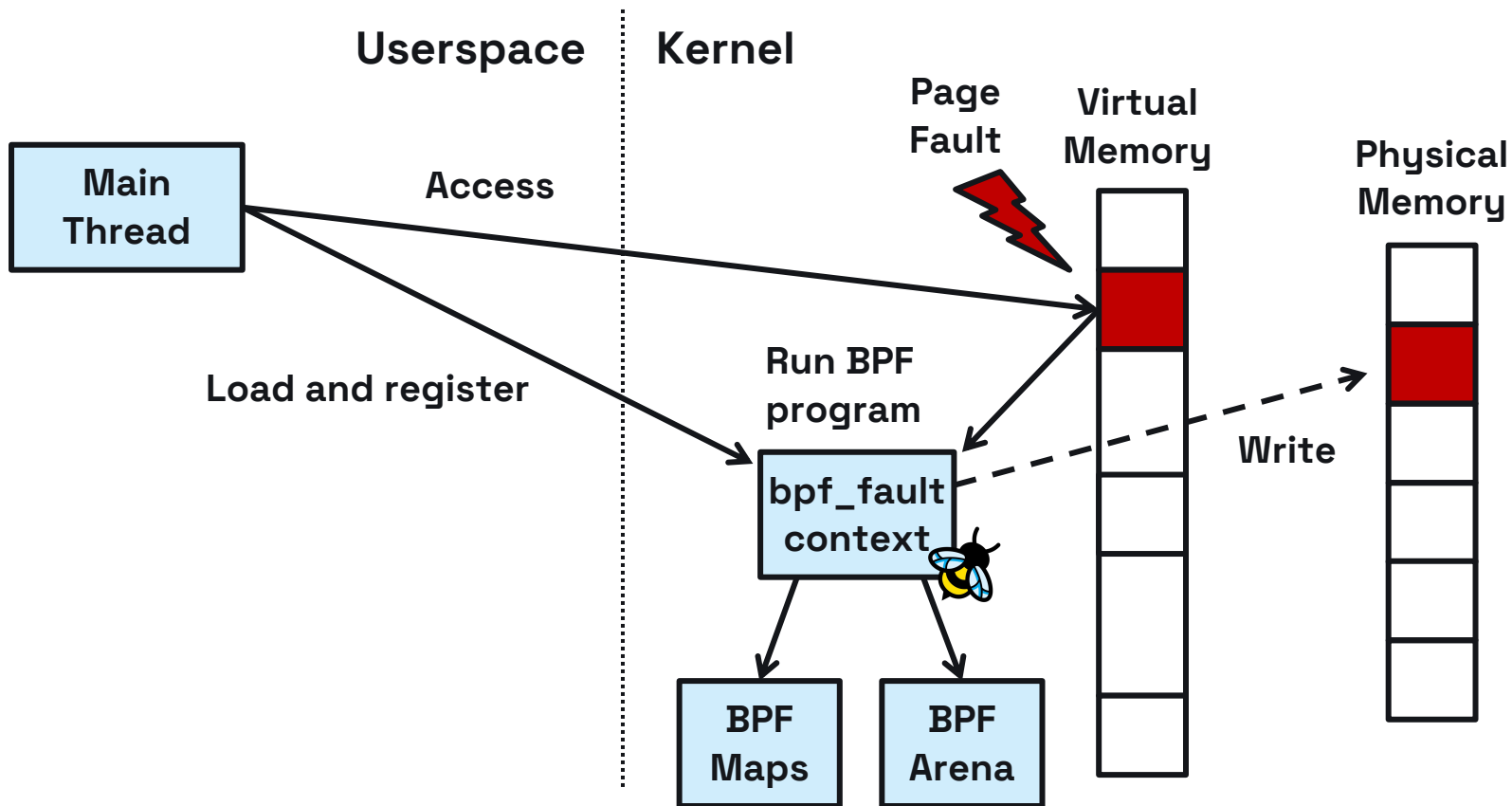
bpf_fault



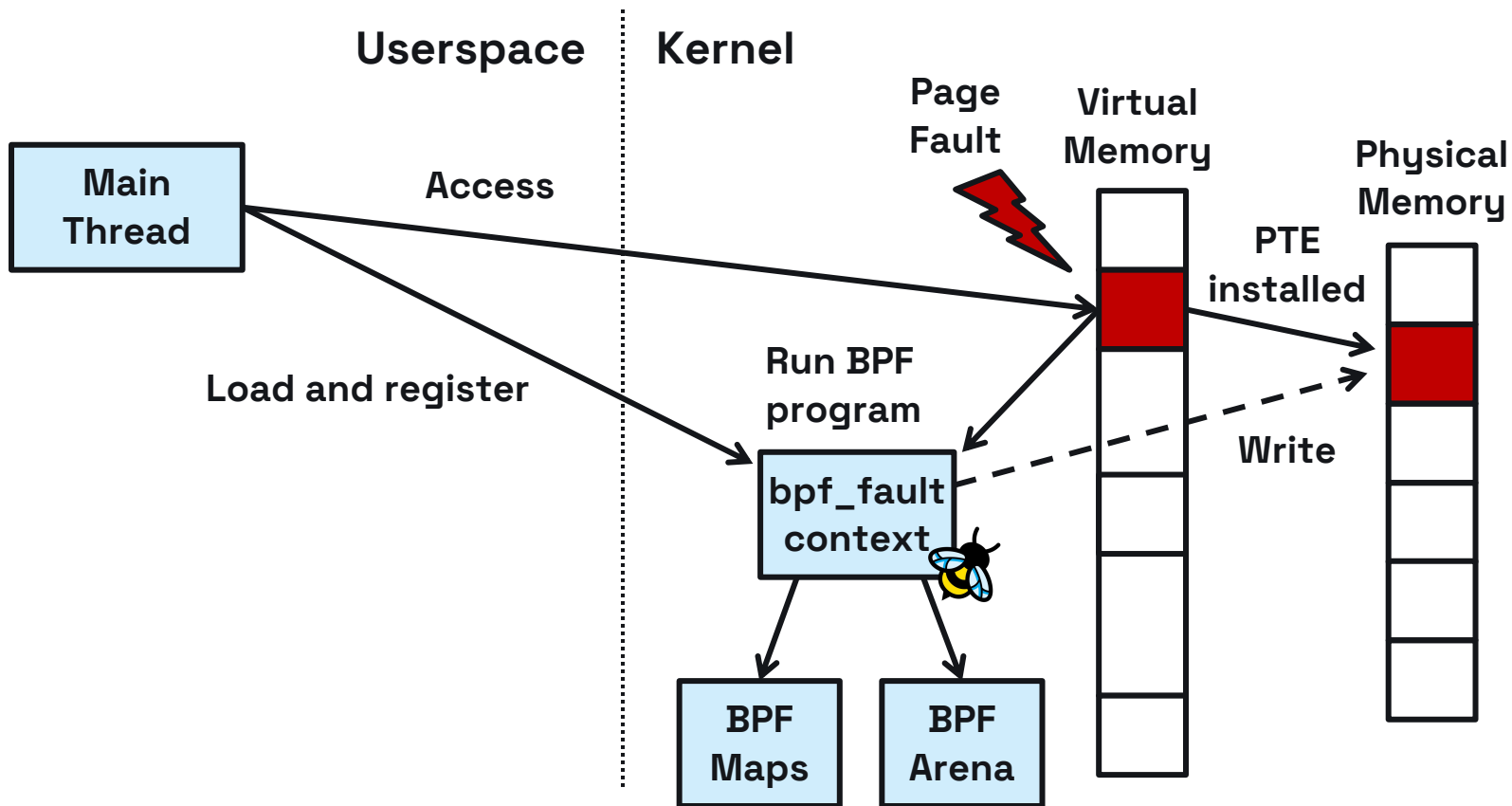
bpf_fault



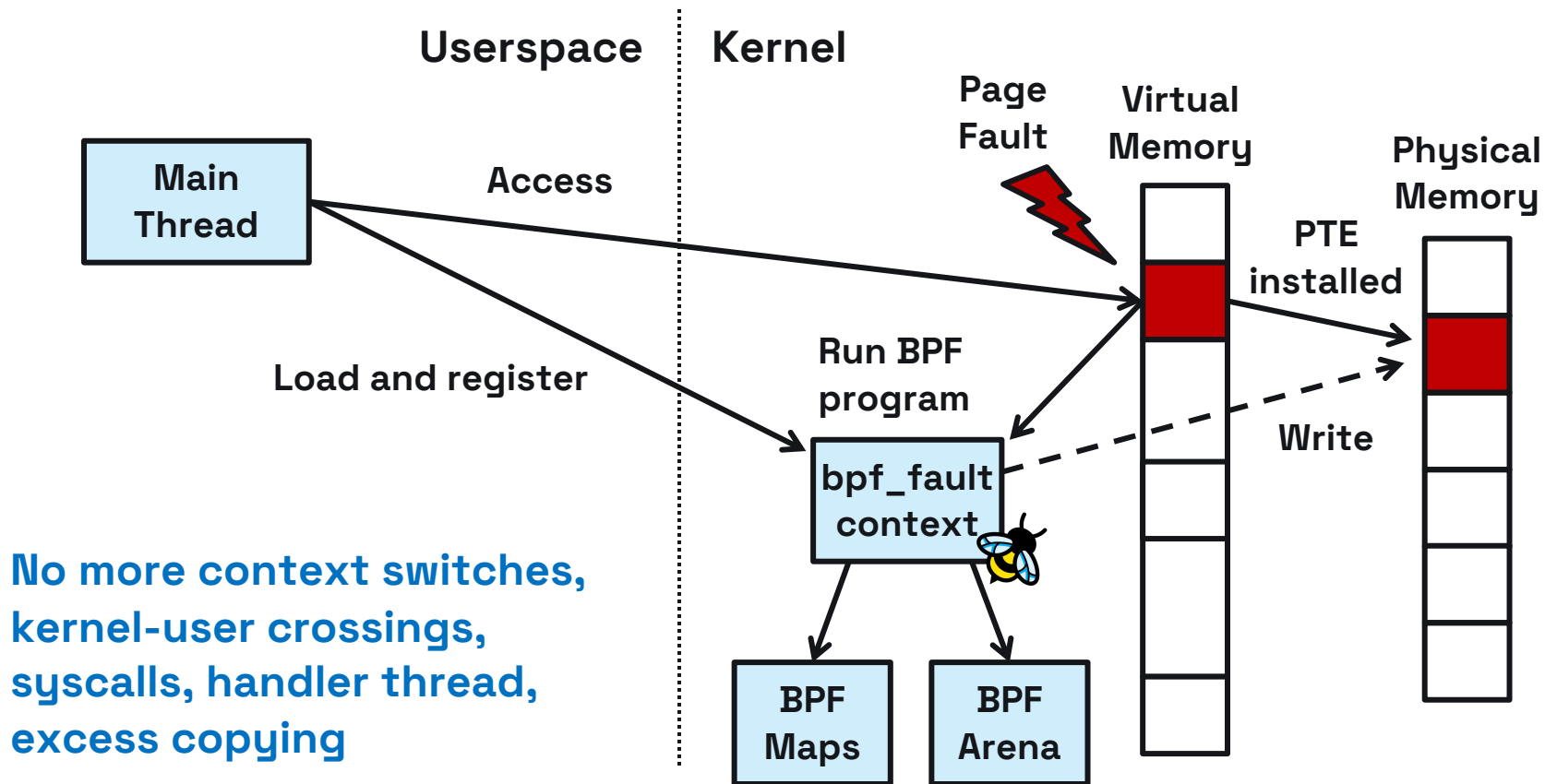
bpf_fault



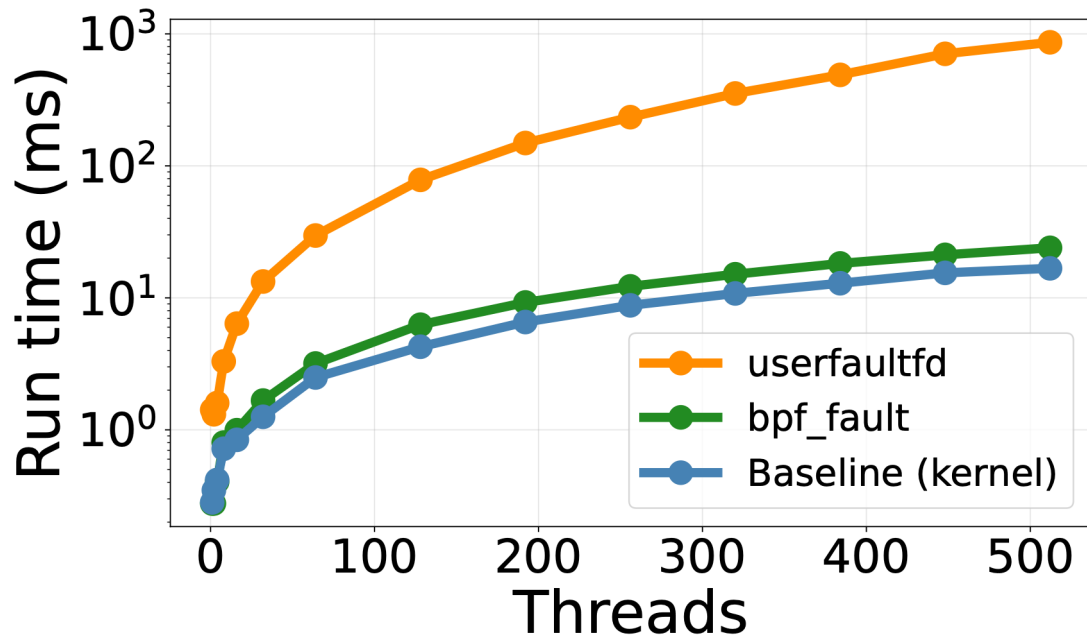
bpf_fault



bpf_fault



bpf_fault scales better and has reduced overhead



6.7× lower latency

compared to uffd, with only 40% overhead relative to baseline.

eBPF Interface

```
/* bpf(BPF_LINK_FAULT_OPS_CMD) commands */
#define BPF_FAULT_WP_ENABLE    (1U << 0)
#define BPF_FAULT_REGISTER    (1U << 1)
#define BPF_FAULT_UNREGISTER  (1U << 2)
#define BPF_FAULT_CLAIM       (1U << 3)

struct bpf_fault_ops_ctx {
    unsigned long address;      /* page-aligned */
    unsigned long real_address; /* exact */
    __u8 fault_type;
};

struct fault_ops {
    int (*handle_missing_fault)(
        struct bpf_fault_ops_ctx *ctx,
        unsigned char *page);
    int (*handle_minor_fault)(...);
    int (*handle_wp_fault)(...);
    int (*handle_fork)(struct bpf_fault_fork_ctx *ctx);
    ...
};
```

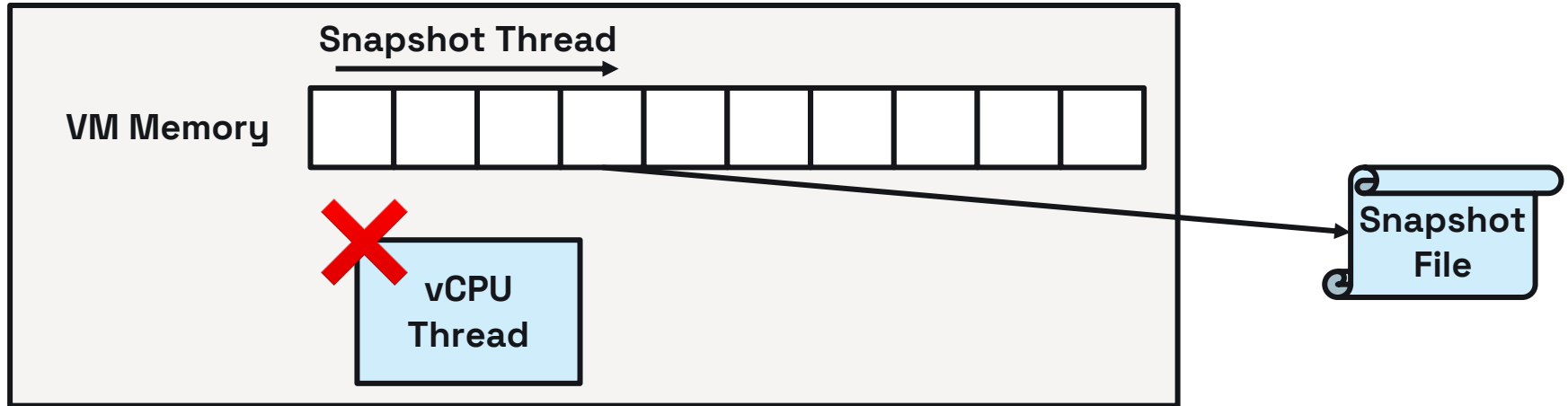
- New `bpf()` commands for region registration, etc.
- `struct_ops` interface
- Events API (à la `uffd`)
- Small verifier change (size associated with `PTR_TO_MEM` to support writing to page ptr)
- Data stored in union with `uffd` VMA fields

APPLICATION 01

VM live snapshots

Stop-the-world snapshots are bad for uptime

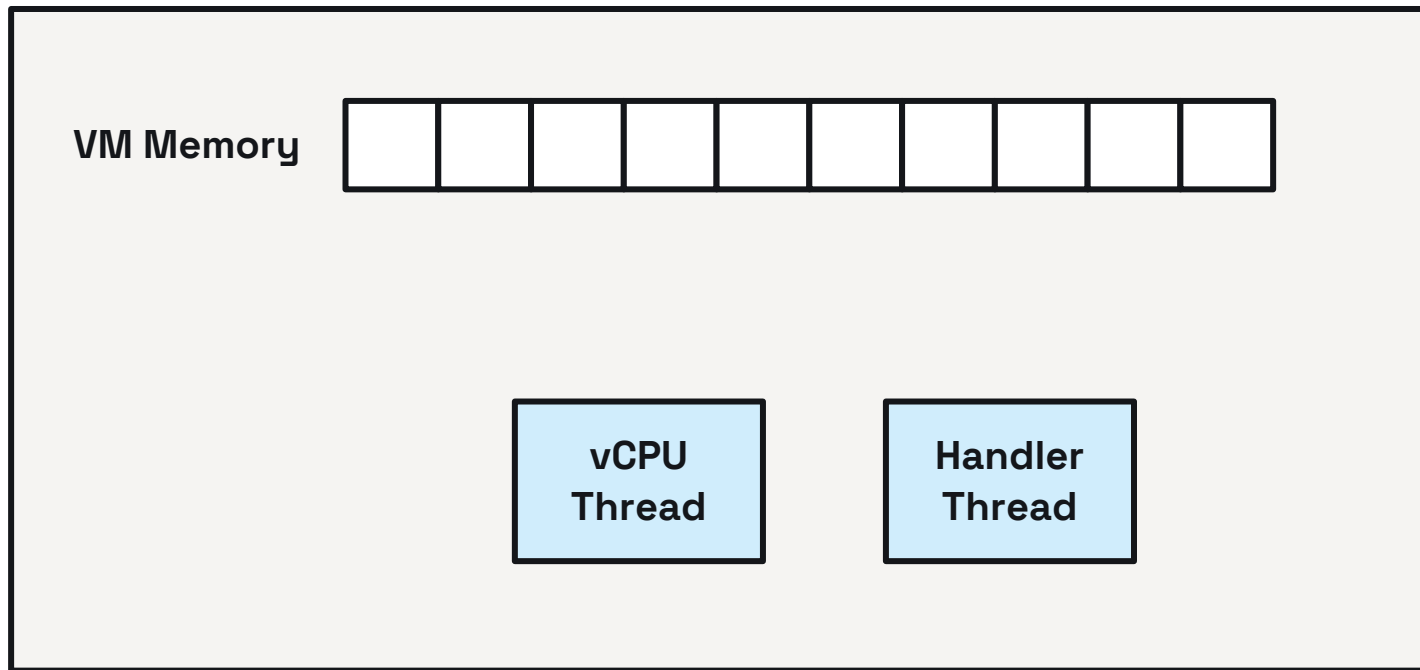
Virtual Machine Manager



VM is fully stopped

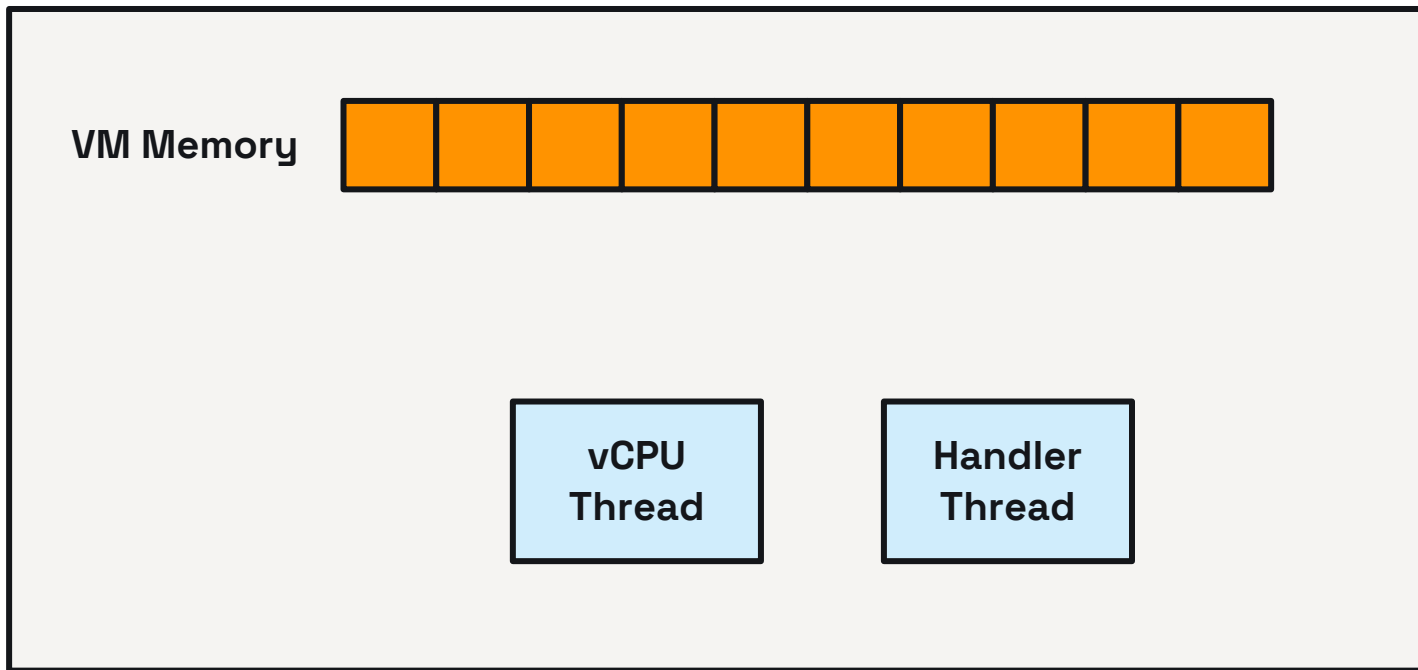
userfaultfd enables live snapshots

Virtual Machine Manager



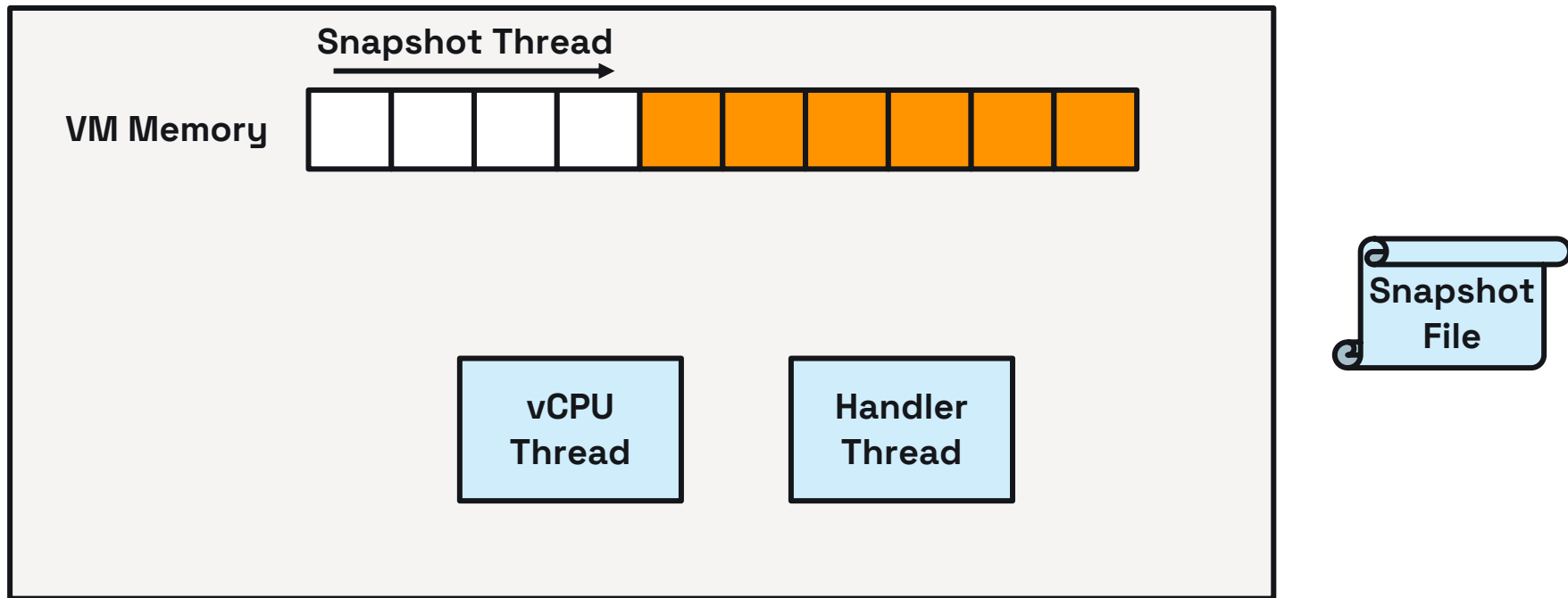
userfaultfd enables live snapshots

Virtual Machine Manager

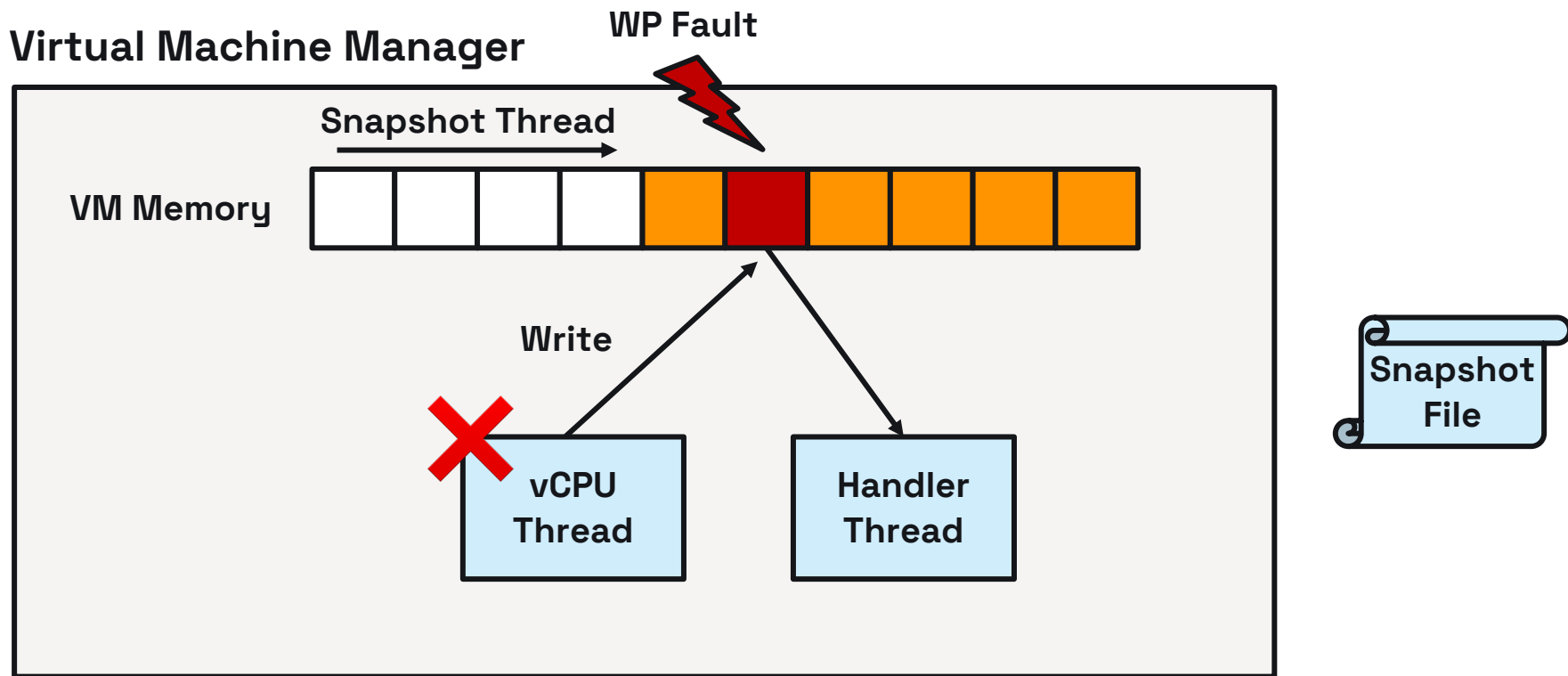


userfaultfd enables live snapshots

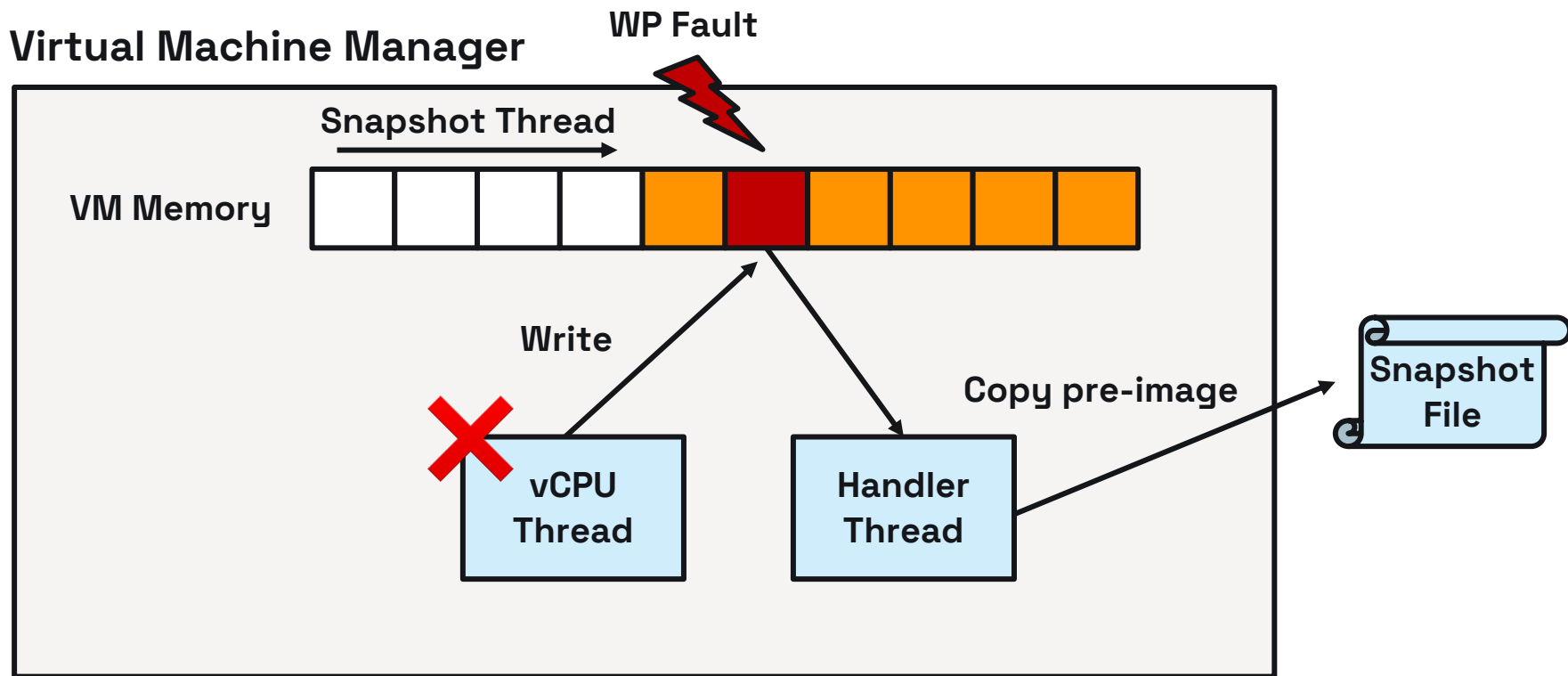
Virtual Machine Manager



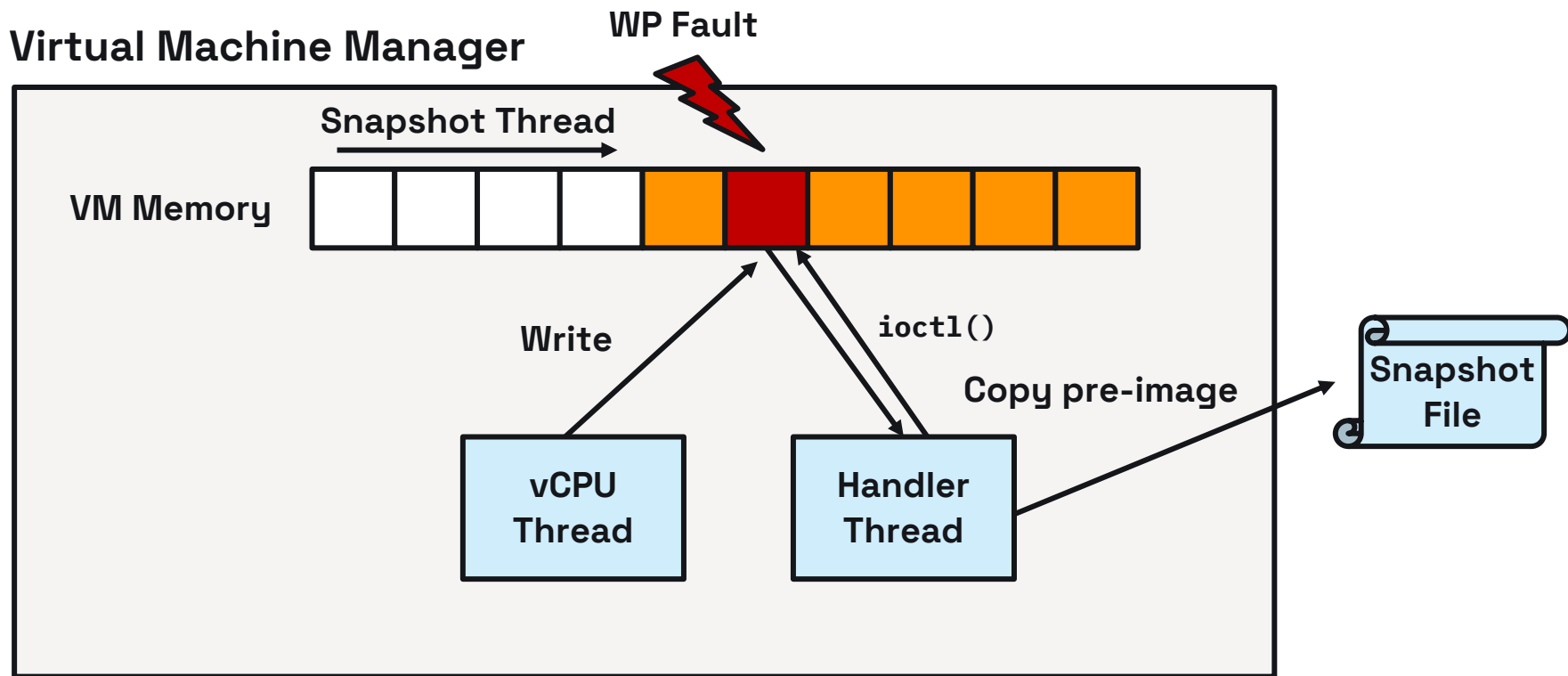
userfaultfd enables live snapshots



userfaultfd enables live snapshots

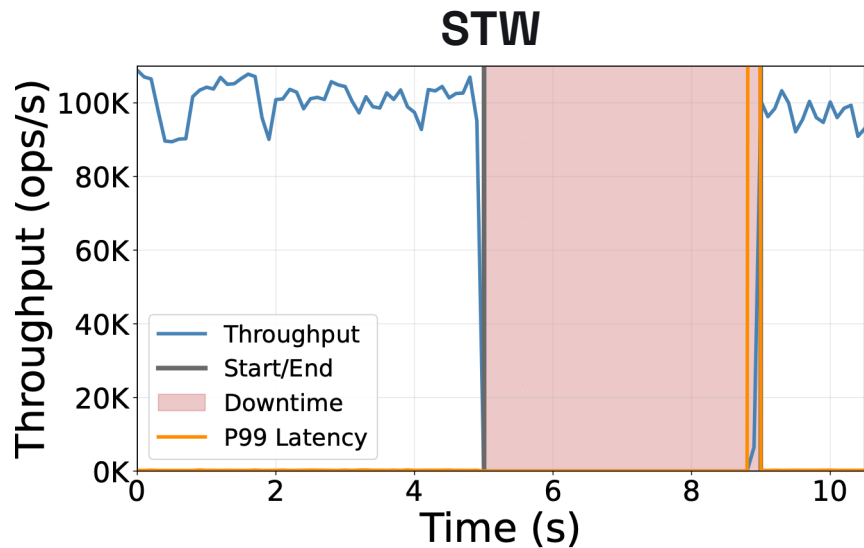


userfaultfd enables live snapshots



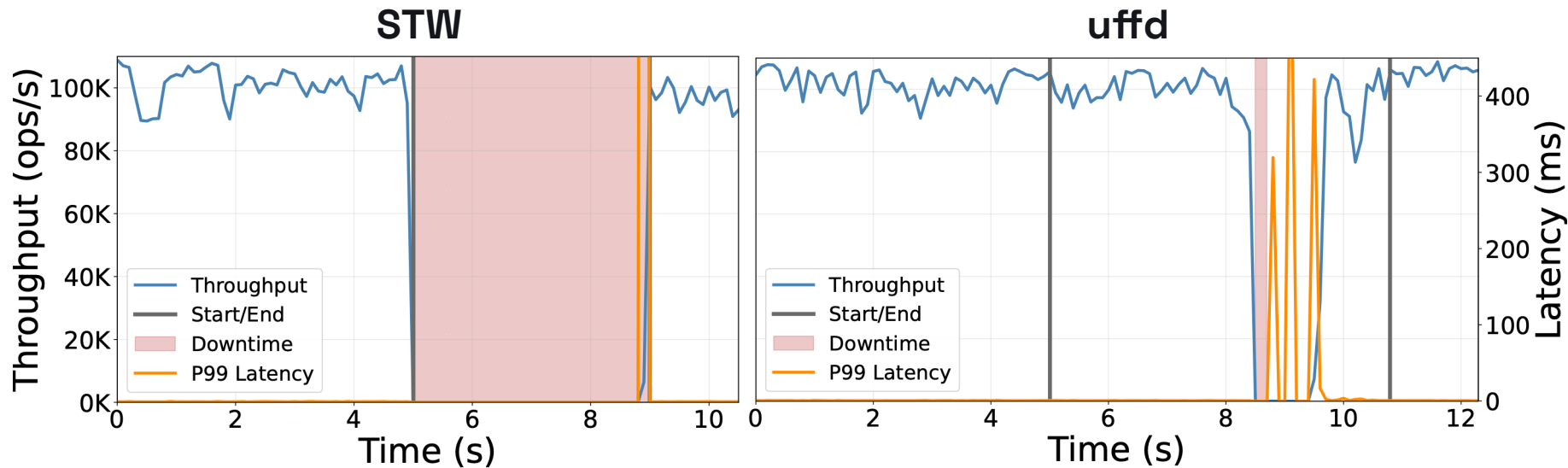
Only blocked during faults

userfaultfd reduces downtime compared to STW



**~4 s of
downtime**

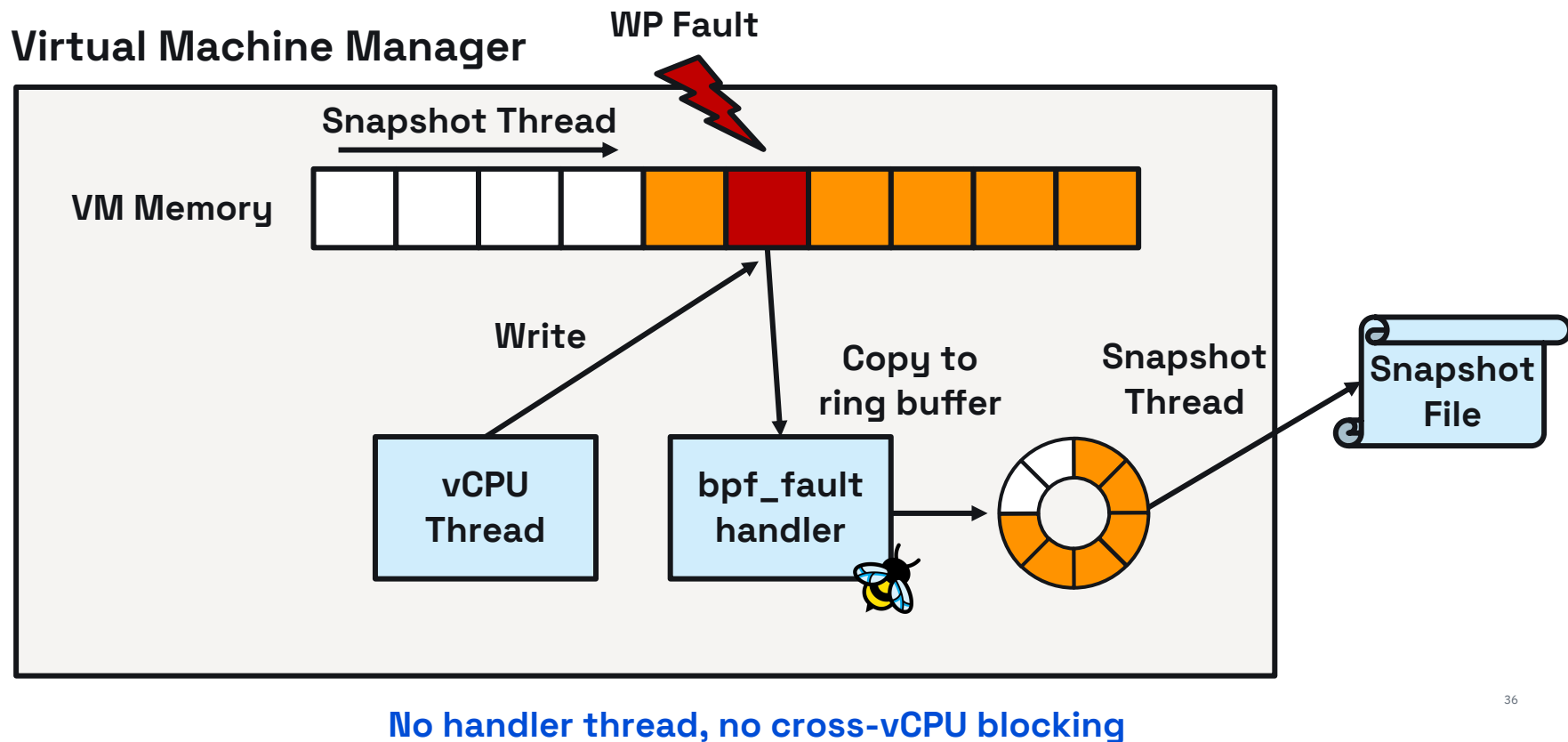
userfaultfd reduces downtime compared to STW



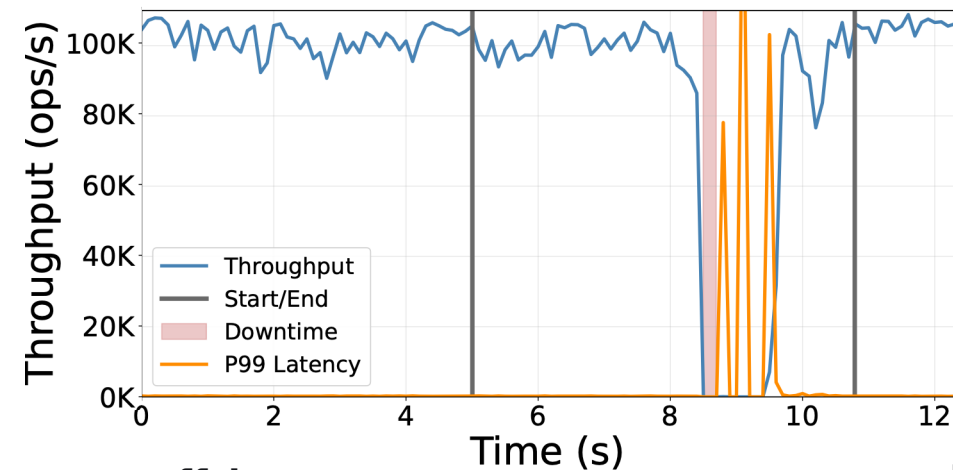
**~4 s of
downtime**

650 ms
repeated p99
latency spikes

bpf_fault enables concurrent vCPU faults



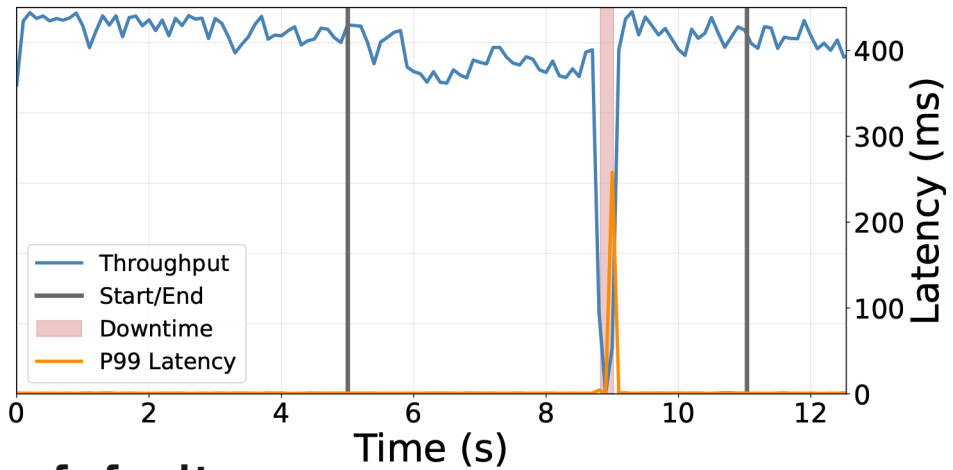
No more latency spikes with bpf_fault



uffd

650 ms

repeated p99
latency spikes

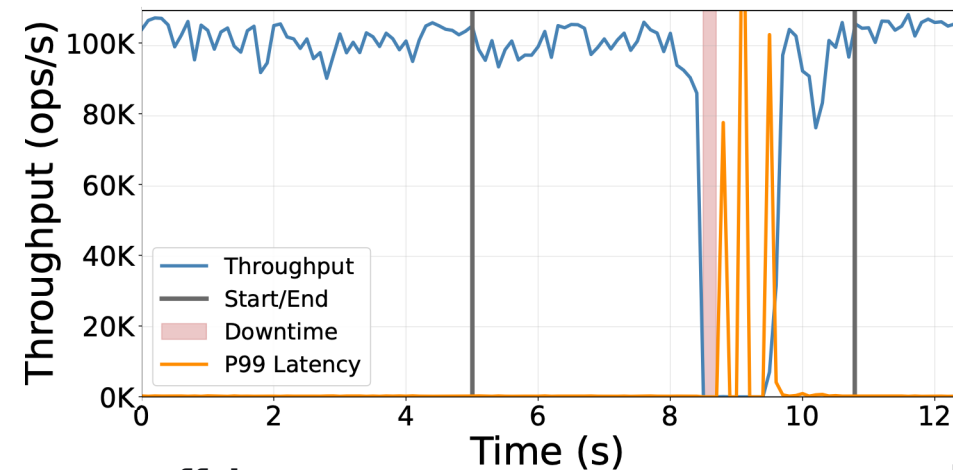


bpf_fault

260 ms

P100 latency,
due to VM pause

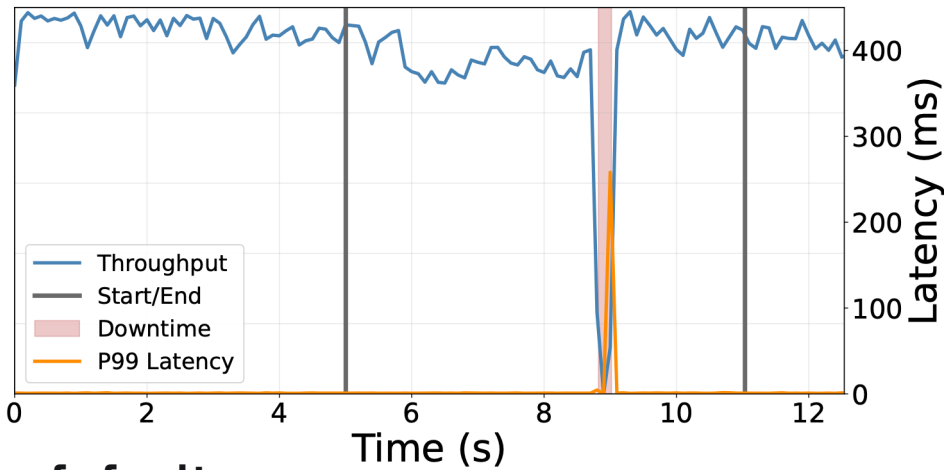
No more latency spikes with bpf_fault



uffd

650 ms

repeated p99
latency spikes



bpf_fault

260 ms

P100 latency,
due to VM pause

2-4x better throughput during snapshot phase for Redis + Memcached
5-11x lower avg. request latency

APPLICATION 02

Lazy dynamic linking

What is dynamic linking?

- The dynamic linker loads an executable and shared libraries
- Has to apply **relocations**: Update binary to account for actual load address

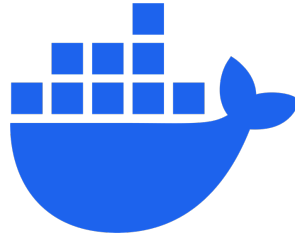
Relative relocations

- Position-independent code can avoid absolute addresses in instructions, but not in data
- Relative relocations:

$$*(\text{base} + \text{r_offset}) = \text{base} + \text{r_addend}$$

Relocations affect startup time and memory usage

- Applied eagerly, triggering COW faults and dirtying the page
- In large binaries:
 - Account for up to 99% of relocations in large binaries
 - 90%+ of dynamic linker startup time, MBs of dirty memory

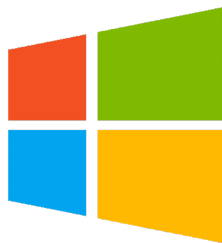


Relative relocations must be applied eagerly

- PLT lazy binding can get away with deferring relocations
 - Functions are in control flow, can execute stub on first use
- No such control flow/execution opportunity for data
 - Only event that triggers on first use is a **page fault**

Other operating systems avoid this problem

- Windows tries to do a common base for objects across processes, per-boot ASLR base
- MacOS applies relocations lazily (“page-in linking”)
 - Changed executable format: Chained fixups (per-page linked list)
- Require changes to executable format, kernel, loader



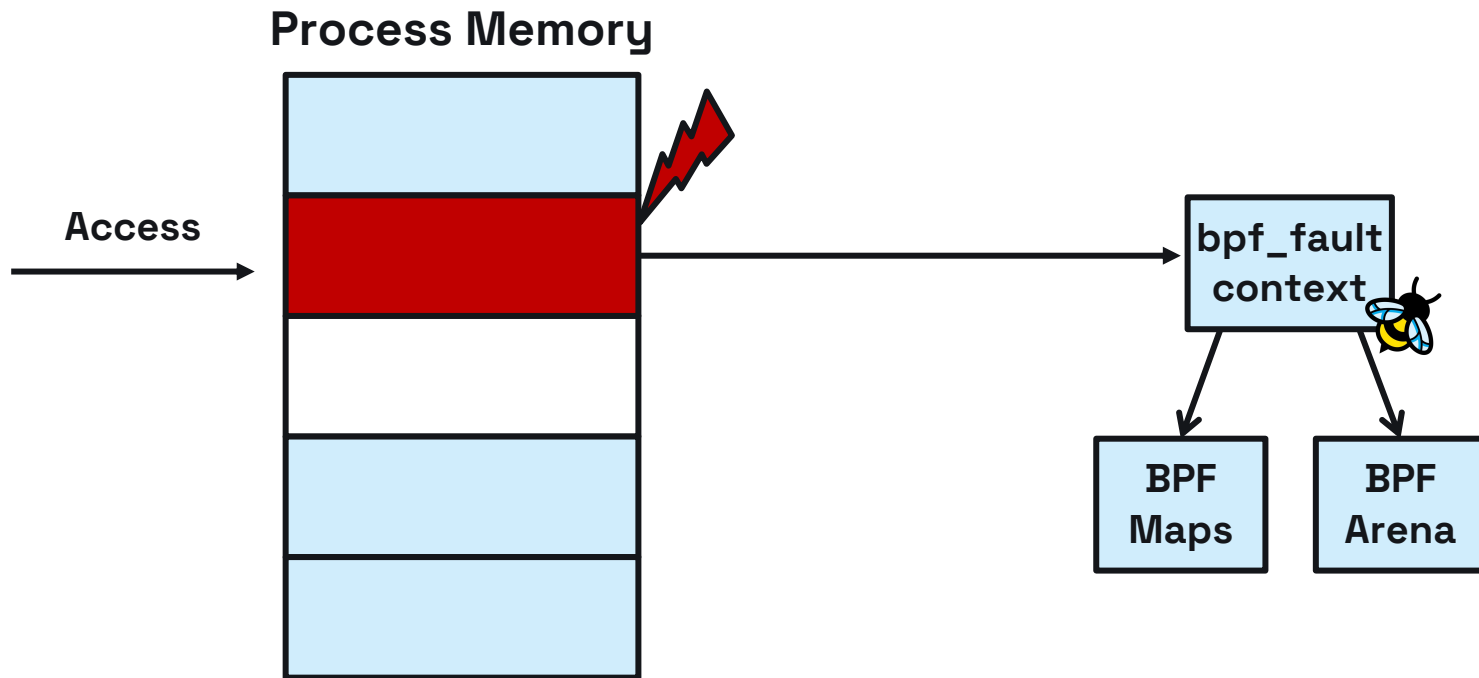
macOS

Other operating systems avoid this problem

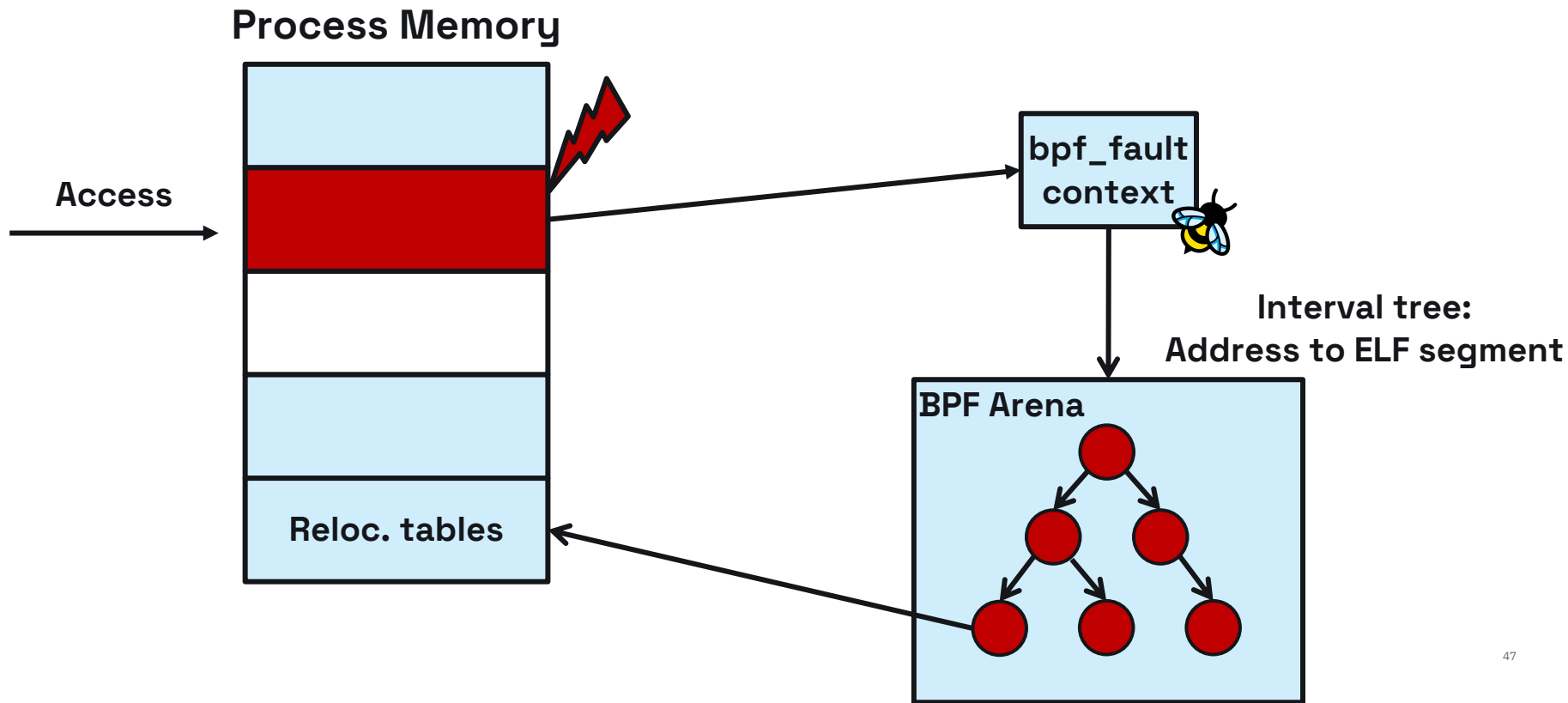
- Windows tries to do a common base for objects across processes, per-boot ASLR base
- MacOS applies relocations lazily (“page-in linking”)
 - Changed executable format: Chained fixups (per-page linked list)
- Require changes to executable format, kernel, loader
- Linux does not:



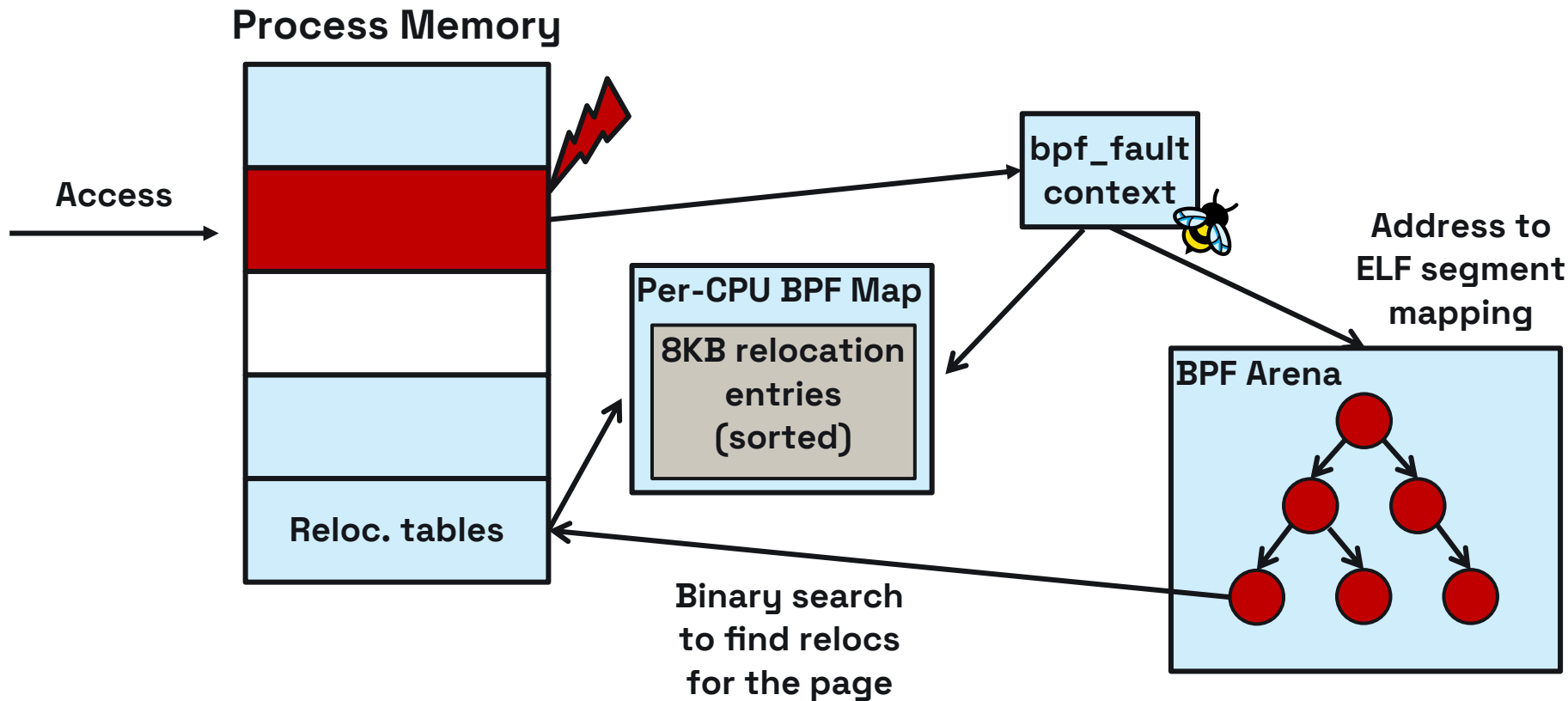
Handle page faults on non-relocated memory



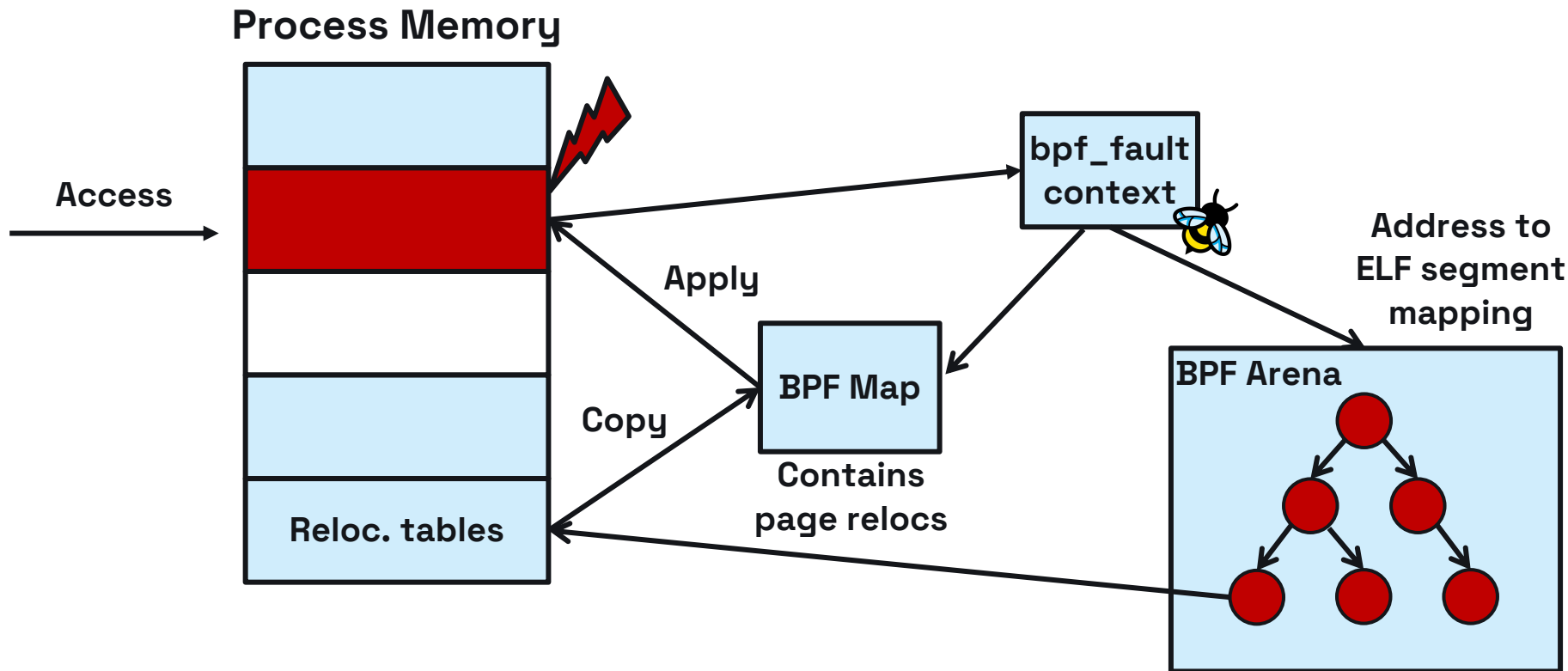
Get the relocations corresponding to the address



Binary search for relocations in the address's page



Write relocations directly to the faulted page



Anything is possible with BPF if you're willing to suffer

- Chrome dirty memory: **200MB -> 100MB**
 - **23% less physical dirty memory** than Chrome's zygote process (Chrome-specific optimization)
- Clang
 - Up to **64% less dirty memory**
 - **7-11% faster compilation**

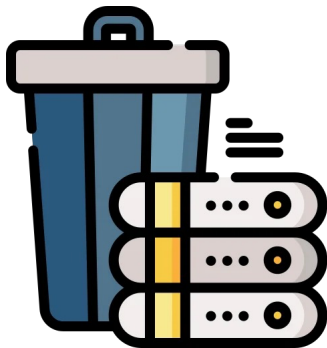
More applications in the paper

GARBAGE COLLECTION

Compaction runs inside the fault handler

Up to 42%

shorter GC pauses in JVM



MEMORY DEBUGGING

Electric fences without a correctness trade-off

4.4× faster

than uffd handler mode



Custom fault handling belongs in the kernel



- **bpf_fault** enables efficient and scalable custom page fault handling
- Integrated with four classes of applications for better performance
- Open source!
- More in paper: Garbage collection, memory debugging, security...

Paper and code:



Tal Zussman

tz2294@columbia.edu

github.com/bpf-fault/bpf-fault