Custom Page Fault Handling With eBPF

Tal Zussman*
Columbia University
tz2294@columbia.edu

Teng Jiang*
Columbia University
tj2488@columbia.edu

Asaf Cidon
Columbia University
asaf.cidon@columbia.edu

ABSTRACT

Traditionally, page faults have been handled by the kernel, with a fixed set of handling routines for different types of faults. However, some applications may benefit from custom page fault handling routines, allowing them to implement advanced functionality, such as more efficient live virtual machine migration and application checkpointing. To this end, Linux introduced the `userfaultfd()` syscall, which allows applications to handle their page faults in userspace. While `userfaultfd()` has proven useful in several applications, we identify some key scalability limitations in its design, which limit both performance and adoption. We propose a system that allows using eBPF programs to handle page faults in-kernel, yielding a simpler and more scalable implementation while also enabling novel use cases, such as accelerating the startup of large position-independent executables like browsers.

CCS CONCEPTS

• **Software and its engineering** → **Memory management**;

KEYWORDS

Operating systems, eBPF, page faults

ACM Reference Format:
Tal Zussman, Teng Jiang, and Asaf Cidon. 2024. Custom Page Fault Handling With eBPF. In *Workshop on eBPF and Kernel Extensions (eBPF '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3672197.3673432

1 BACKGROUND

While page faults are traditionally handled by the kernel, in some cases it is beneficial to let applications customize page fault handling. For example, virtual machine (VM) live migration allows virtual clusters to migrate VMs across physical hosts with minimal disruption to the guest [5]. It is an important capability, commonly used for VM/host software upgrades, load balancing, hardware failure handling, and scheduled maintenance [25]. When using the post-copy migration strategy, most of the VM's memory gets copied on-demand, thereby minimally impacting the application [13, 15]. A similar technique can be used for checkpoint-restore-in-userspace (CRIU), which saves a process's state to disk and restores it at a later point [10]. When a process is restored, rather than copying its entire state to its address space, relevant pages can be copied

*Equal contribution

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
eBPF '24, August 4–8, 2024, Sydney, NSW, Australia
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0712-4/24/08.
https://doi.org/10.1145/3672197.3673432

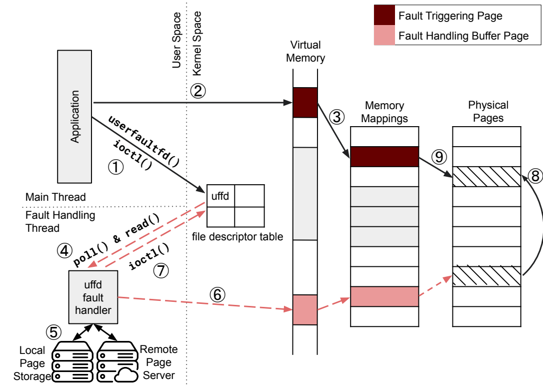

Figure 1: `userfaultfd()` workflow.

on-demand, reducing start-up time. While such functionality could be supported through significant kernel modifications [13], a more general-purpose solution is desirable.

To solve this problem, Linux introduced the `userfaultfd()` syscall, allowing applications to handle page faults in userspace with a dedicated fault-handling thread [2, 17]. We use the CRIU application to illustrate a `userfaultfd()` workflow (Figure 1).

Assume we want to restore an application to a checkpointed state with CRIU (i.e. its state is stored on a local or remote disk). In this case, the CRIU client sets up an initial subset of the application's pages. The remaining pages aren't initialized yet, and they only get copied when the application tries to access them, requiring a custom page fault handler. The CRIU client then calls `userfaultfd()`, creating a new uffd (userfault file descriptor), and uses `ioctl()` to register the regions of virtual memory (the application's remaining pages) it should handle (①). After creating a fault-handling thread that polls on the uffd, the application can safely resume execution. When the application attempts to access an unmapped page (② and ③), a page fault is raised. If the relevant page is in a `userfaultfd()`-registered region, the kernel marks the uffd as ready, waking up the fault-handling thread, which then reads from the uffd (④). For CRIU, the fault-handling thread will read the relevant data from the application's (local or remote) checkpoint (⑤) into a local buffer (⑥). The thread then submits a `userfaultfd()` command using `ioctl()` (⑦), which atomically copies the data into the application's address space, resolving the fault (⑧, ⑨).

In addition to CRIU, `userfaultfd()` has seen adoption in a number of large projects, along with more experimental work [18, 22, 23, 26–28]. `userfaultfd()` is used by QEMU for VM post-copy migration, while Firecracker uses it to lazily restore microVM memory from a snapshot [1]. The Android Runtime's garbage collector uses `userfaultfd()` in its compaction phase to track page accesses [3, 12]. Additionally, the authors of `userfaultfd()` have identified several additional potential use cases, such as distributed shared memory, language runtimes, and JIT compilers,

along with adding support for handling write-protect and minor faults [2, 6, 7, 24]. Finally, we believe that there exist additional use cases for custom page fault handling which cannot utilize `userfaultfd()` due to its limitations, such as lazily resolving data relocations for position-independent executables in the dynamic linker, yielding faster program start-up [4] (see §2).

2 LIMITATIONS OF USERSPACE FAULT HANDLING

`userfaultfd()`'s design allows applications to fully customize page fault handling, with practically no limitations on the fault-handling routine. However, its design also incurs significant costs. In this section, we describe three limitations of `userfaultfd()`: scalability, applicability, and security.

Scalability. In multi-threaded applications, typically each thread's page faults are handled in-kernel by the respective thread. However, for applications using `userfaultfd()`, there is only one fault-handling thread, which can become a performance bottleneck. To illustrate this, we run an experiment with and without `userfaultfd()`: we set up a pool of threads, each of which is allocated 50 pages of anonymous memory, and access each of those pages, generating 50 page faults per thread. We then fill each page with a fixed value, either in the handling routine (using `userfaultfd()`) or in the thread loop (default fault handling). As shown in Figure 2, as the number of threads increases, `userfaultfd()` takes significantly longer to handle page faults and scales worse than the standard kernel handling.

Additionally, each `userfaultfd()` fault-handling routine requires at least three syscall invocations: `poll()`, `read()`, and `ioctl()`. Each of these syscalls requires user-kernel crossings, which add overhead for each page fault, in addition to the cost of context switching to the fault-handling thread. Finally, `userfaultfd()`'s design may lead to unnecessarily copying data between userspace and kernel space. For example, in the VM migration use case, data read from the network is copied to userspace through `recv()`, and then copied back to the kernel as part of the `userfaultfd()` resolution code. Since the data is already present in the kernel, this overhead is unnecessary.

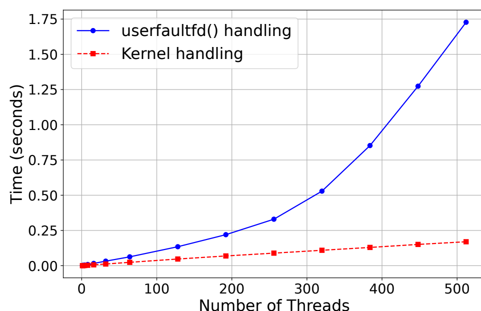

Figure 2: Latency to handle 50 page faults per thread.

Applicability. The use of a dedicated fault-handling thread limits which systems can use `userfaultfd()`. Specifically, applications that fork may not be able to use `userfaultfd()`, as forking in a multi-threaded environment only clones the thread that calls `fork()`, potentially leading to incorrect behavior. This effectively prevents `userfaultfd()` from being used in interchangeable libraries, such as the dynamic linker or garbage collectors, which cannot restrict application behavior in such a way.

Security. `userfaultfd()` has been used in a number of kernel exploits [8, 14, 16, 20, 21], many of which have taken advantage of its ability to indefinitely block kernel execution at a specific point. While mitigations have been developed [8], container runtimes such as Docker have blocked its usage in their default configurations due to these security concerns [11], further limiting its applicability.

3 DESIGN

In order to mitigate some of the downsides of `userfaultfd()`, we propose an **eBPF-based system** to register custom page fault handlers in-kernel. This system requires two key modifications to the kernel, described below.

Per-VMA eBPF programs. Each eBPF fault-handling program must be associated with a range of the process's address space. The relevant kernel data structure is the `struct vm_area_struct`, which represents a contiguous virtual memory area (VMA). As such, we add support for per-VMA eBPF programs, similar to the kernel's support for per-cgroup eBPF programs [19]. After loading and verifying the eBPF program, the application attaches it to a specific address range. The kernel then translates that address range to a VMA (or series of VMAs), and associates the eBPF program with those VMAs. If the address range starts or ends within a VMA, the kernel will split the VMA, as is done with `userfaultfd()`.

Fault-handling modifications. We modify the kernel's page fault-handling routine to check if an eBPF program is attached to the relevant VMA. If so, we run the eBPF program, providing metadata about the fault, along with access to a newly allocated page to fill with the desired contents. After the eBPF program executes, we resolve the fault by setting the relevant memory mappings to point to the new page. This removes the need for an additional memory copy, as is done in `userfaultfd()`, which fills the page in userspace and then copies it into the kernel. We envision this design enabling zero-copy page faults, with data read from the disk or network written directly to the relevant page.

For simple fault-handling routines, the aforementioned kernel changes should be sufficient. However, for more complex applications such as VM migration which require reading data from the network or disk, we envision adding eBPF helpers to support such operations, along with building on the existing support for sleepable eBPF programs [9].

We believe that this implementation removes the need for an additional fault-handling thread and reduces unnecessary kernel crossings or data copying. While eBPF may somewhat limit the flexibility of the custom fault handlers, we believe that eBPF is mature enough to handle interesting and complex use cases. Additionally, the eBPF verifier could be used to limit the operations used in the handling routine, such as sleeping indefinitely, potentially addressing the security concerns that plague `userfaultfd()`.

4 ACKNOWLEDGMENTS

This work was supported by Intel and IBM, and NSF awards CNS-2143868 and CNS-2104292. Tal Zussman was supported by NSF award DGE-2036197.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[2] Andrea Arcangeli. 2016. Userland Page Faults and Beyond: Why How and What's Next. https://www.linux-kvm.org/images/1/10/01Wed-1415-LinuxCON-aarcangeli-userfaultfd.pdf. (2016).

[3] Seang Chau. 2022. Android 13 is in AOSP. https://android-developers.googleblog.com/2022/08/android-13-is-in-aosp.html. (2022).

[4] Chromium [n. d.]. Native Relocations. https://chromium.googlesource.com/chromium/src/+/master/docs/native_relocations.md#Linux-Android-Relocations-ELF-Format. ([n. d.]).

[5] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)*. USENIX Association, USA, 273–286.

[6] Jonathan Corbet. 2017. The next steps for userfaultfd(). https://lwn.net/Articles/718198. (2017).

[7] Jonathan Corbet. 2019. Write-protect for userfaultfd(). https://lwn.net/Articles/787308. (2019).

[8] Jonathan Corbet. 2020. Blocking userfaultfd() kernel-fault handling. https://lwn.net/Articles/819834/. (2020).

[9] Jonathan Corbet. 2020. Sleepable BPF programs. https://lwn.net/Articles/825415/. (2020).

[10] CRIU Project 2019. CRIU. https://criu.org. (2019).

[11] Docker [n. d.]. Seccomp security profiles for Docker. ([n. d.]). https://docs.docker.com/engine/security/seccomp.

[12] Lokesh Gidra, Hans-J. Boehm, and Joel Fernandes. 2020. Utilizing the Linux Userfaultfd System Call in a Compaction Phase of a Garbage Collection Process. *Technical Disclosure Commons* (12 10 2020). https://www.tdcommons.org/dpubs_series/3671

[13] Michael Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-copy live migration of virtual machines. *Operating Systems Review* 43 (07 2009), 14–26. https://doi.org/10.1145/1618525.1618528

[14] Jann Horn. 2016. CVE-2016-4557: Linux: UAF via double-fdput() in bpf(BPF_PROG_LOAD) error path. https://bugs.chromium.org/p/project-zero/issues/detail?id%3D808. (2016).

[15] Jaeseong Im, Jongyul Kim, Youngjin Kwon, and Seungryoul Maeng. 2022. On-Demand Virtualization for Post-Copy OS Migration in Bare-Metal Cloud. *IEEE Transactions on Cloud Computing* PP (01 2022), 1–1. https://doi.org/10.1109/TCC.2022.3179485

[16] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. DirtyCred: Escalating Privilege in Linux Kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1963–1976. https://doi.org/10.1145/3548606.3560585

[17] Linux Documentation [n. d.]. Userfaultfd. https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt. ([n. d.]).

[18] Robert Lyerly, Xiaoguang Wang, and Binoy Ravindran. 2020. Dynamic and Secure Memory Transformation in Userspace. In *Computer Security – ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg, 237–256. https://doi.org/10.1007/978-3-030-58951-6_12

[19] Daniel Mack. 2016. Per-cgroup BPF Programs. https://lore.kernel.org/all/1479916350-28462-1-git-send-email-daniel@zonque.org/T/. (2016).

[20] Vitaly Nikolenko. 2016. CVE-2016-6187: Exploiting Linux kernel heap off-by-one. https://duasynt.com/blog/cve-2016-6187-heap-off-by-one-exploit. (2016).

[21] Vitaly Nikolenko. 2018. Linux Kernel universal heap spray. https://duasynt.com/blog/linux-kernel-heap-spray. (2018).

[22] Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. 2019. UMap: Enabling Application-driven Optimizations for Page Management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 71–78. https://doi.org/10.1109/MCHPC49590.2019.00017

[23] Ivy B. Peng, Maya B. Gokhale, Karim Youssef, Keita Iwabuchi, and Roger Pearce. 2022. Enabling Scalable and Extensible Memory-Mapped Datastores in Userspace. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 866–877. https://doi.org/10.1109/TPDS.2021.3086302

[24] Mike Rapoport. 2017. Userfaultfd: Post-copy VM Migration and Beyond. https://blog.linuxplumbersconf.org/2017/ocw/system/presentations/4699/original/userfaultfd_%20post-copy%20VM%20migration%20and%20beyond.pdf. (2017).

[25] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. 2018. VM Live Migration At Scale. *SIGPLAN Not.* 53, 3 (Mar 2018), 45–56. https://doi.org/10.1145/3296975.3186415

[26] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. 2022. zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 431–445. https://www.usenix.org/conference/osdi22/presentation/stamler

[27] Ville Tuulos. 2016. Querying Data in Amazon S3 Directly with User-Space Page Fault Handling. https://tech.nextroll.com/blog/data/2016/11/29/traildb-mmap-s3.html. (2016).

[28] Kan Zhong, Wenlin Cui, Xin Chen, Qiao Li, Zhe Yang, Youyou Lu, Xiaodan Yan, Siwei Luo, Qizhao Yuan, and Keji Huang. 2021. Revisiting Swapping in User-Space With Lightweight Threading. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42 (2021), 4205–4218. https://api.semanticscholar.org/CorpusID:236493177