Technical Disclosure Commons

Defensive Publications Series

October 2020

Utilizing the Linux Userfaultfd System Call in a Compaction Phase of a Garbage Collection Process

Lokesh Gidra

Hans-J. Boehm

Joel Fernandes

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation

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

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

Utilizing the Linux *Userfaultfd* System Call in a Compaction Phase of a Garbage Collection Process

Abstract:

This publication describes techniques for utilizing the Linux *userfaultfd* system call in a garbage collection process performed concurrently with the execution of application threads (mutators) in a software application. During the garbage collection process, a stop-the-world pause occurs where currently mapped physical pages of a heap are moved to a temporary location (*e.g.*, temp-space) and a new memory range of the heap is registered with *userfaultfd*. During a concurrent compaction phase of the garbage collection process, if a mutator accesses an area (*e.g.*, a to-space page) in the heap that has not yet been processed by the garbage collector thread, and thus does not have a page allocated, the mutator will receive a SIGBUS signal (bus error) indicating a page fault. Responsive to receiving the registered page fault, a page buffer (*e.g.*, 4KB page buffer) is created. All the reachable objects that should be located on the missing page are copied to the page buffer and the references inside these objects are updated to the corresponding new addresses. Finally, the *userfaultfd* input/output control (*ioctl*) system call is invoked by the user space to hand over the page buffer to the kernel, including an indication of the page to make visible on the faulting address. In response, the kernel can copy the contents of the page buffer to a page and map that page.

Keywords:

automatic dynamic memory management, garbage collection, garbage collector, GC, computing device, application, user space, kernel, Java, Linux, *userfaultfd*, *mremap, mprotect,* system call, syscall, input/output control, *ioctl*, object, marking, compaction, concurrent

Background:

Garbage collection is an automatic dynamic memory management process by which a garbage collector algorithm (the "garbage collector") reclaims memory occupied by objects that are no longer in use by a running software application. The reclamation of memory may include moving reachable objects within the heap from a first memory range to a second memory range concurrently with the execution of application threads (also referred to as "mutators"). Copying objects while application threads are accessing, and possibly updating, the objects can cause data consistency issues.

In aspects, a garbage collection process includes a concurrent compaction phase where the garbage collector compacts (relocates) the objects that remain in the heap. To implement such a concurrent compaction phase, a short stop-the-world (STW) pause is executed where all entry points into the heap, typically referred to as "garbage-collection roots" (GC-roots), are updated with the new (moved) location of the objects that the GC-roots were originally mapped to. Thereafter, the application threads are resumed and compaction is performed concurrently. Such a concurrent compaction phase permits live objects to be moved to their new location in the compacted world without other programs and applications attempting to access or update them, as this could lead to data-consistency issues.

Garbage collectors may undertake a number of different compaction approaches to avoid such data consistency issues during the compaction phase. In a first compaction approach, the garbage collector requires mutators to perform garbage collection activity on reads and/or writes to live objects. To do this, the garbage collector inserts read/write barriers into the application's code for every reference load. This is done so that when the garbage collection is performed, if a mutator accesses objects in the heap that are being copied and compacted by the garbage collector,

then such access is intercepted by a read/write barrier. Utilization of a read/write barrier in such a fashion enables the garbage collector to check and ensure that consistent data is accessed by the mutator. The "Concurrent Copying (CC)" garbage collector implemented in Android version 10, uses a read/write barrier to ensure that the mutator always sees the copied object during garbage collection, thereby eliminating data consistency concerns.

The second compaction approach utilizes page-level memory protection to intercept accesses to the heap. For example, the to-space is protected using *mprotect*, a system call for controlling memory protections. When *mprotect* is activated, before the objects are copied and/or moved, the memory range of the heap is protected so that access to the to-space is protected. As a result, once the mutators resume, on accessing the to-space, they receive a SIGSEGV signal (segmentation fault signal) that triggers a fault handler (*e.g.*, page fault signal handler). A consistent view of the memory area being accessed can be created in the fault handler before making the memory area visible to the application. For example, the fault handler may copy all the objects in the faulted page after the compaction, update all the references in them, and then make the page accessible.

Description:

This publication describes techniques for utilizing the Linux *userfaultfd* system call in a garbage collection process implemented by a garbage collector algorithm that performs garbage collection concurrently with executing application threads (mutators). The process includes a number of different phases that may be performed concurrently with the execution of mutators. The phases include a concurrent marking phase, a concurrent computation phase, a stop-the-world setup phase, and a concurrent compaction phase, as illustrated in Fig. 1 below.

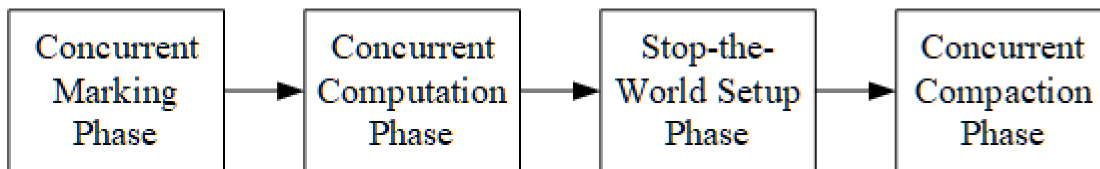

Fig. 1

In the marking phase, a garbage collector algorithm (the "garbage collector") determines the usefulness of objects within the heap by attempting to reach each object in the heap. If an object is reachable by the garbage collector, then that object is considered live and useful; whereas if the object is unreachable, it is considered dead. The garbage collector marks reachable objects by populating a mark bitmap. The concurrent marking of objects can be performed using any garbage collector algorithm that generates a mark bitmap that has a set bit for every reachable object. For example, the Concurrent Mark Sweep (CMS) garbage collector implemented in the Android runtime may be used for the concurrent marking of dead objects in a concurrent marking phase.

Linux introduced the *userfaultfd* system call in kernel version 4.19. The *userfaultfd* system call allows the user space to deal with page faults (*e.g.*, page-missing page faults) in a registered memory range. Once the memory range is registered with *userfaultfd*, page-fault handling is delegated to a user-space application through a *userfaultfd* notification. The *userfaultfd* notification may be received either through a SIGBUS (bus error) fault handler or by reading from the file descriptor registered with *userfaultfd*. When the user-space application receives control and notification of page fault, the user-space application creates content that should be mapped on the page of the faulting address, resolving the page fault. The user-space application then hands the content over to the kernel.

The concurrent computation phase occurs after the concurrent marking phase. In the concurrent computation phase, information gathered in the concurrent marking phase (*e.g.*, the

mark bitmap) is used to pre-compute information for every object, including the new location of each object in the compacted world. The garbage collector iterates over the mark bitmap and generates data enabling the implementation of a function that works in constant time, taking (as an input) the old address of a reachable object, and outputting the new address of the object. The new address is the location where the object will be copied for compaction in the later compaction phase. An example concurrent computation phase includes two steps. First, the from-space offset, from where the live objects must be copied, is computed for every to-space page. Computing the from-space offset ensures that the mutators and the garbage collector thread, while processing a to-space page, can find a source offset in O(1) time. Second, an offset-vector defining the offset between the old address and the new address is determined. This offset-vector, together with the mark bitmap, can be used to define the to-space address of any object in O(1) time.

After the concurrent computation phase, the stop-the-world (STW) setup phase occurs. In the STW setup phase, a STW pause occurs where currently mapped physical pages of the heap (*e.g.*, from-space) are moved to a temporary location (*e.g.*, temp-space) and the new memory range of the heap is registered with *userfaultfd*. By so moving the pages, the original heap range will be entirely empty. As a result, once the execution of mutators resumes and the mutators attempt to access a location with a page missing, the mutators will receive a page fault.

The operation of moving the physical pages of the heap to the temporary location can be performed using the *mremap* system call, whose cost is linear in the number of pages to move. Given the high response sensitivity of some computing devices, a stop-the-world pause proportional to the heap size in length may cause janks. To avoid janks, *mremap* can be optimized by moving the page-table entries inside the kernel at Page Middle Directory (PMD) entry level instead of at the Page Table Entry (PTE) level. This ensures that instead of moving 4KB at a time,

2MB memory is moved, thereby significantly reducing the number of iterations required. This can be further scaled up to work at Page Upper Directory (PUD) level or Page Global Directory (PGD) level if the target is to move multi-gigabyte or larger heaps. The only requirement is to ensure that heap and temporary memory ranges are appropriately aligned.

In parallel with moving the pages and registering the memory range of the heap, the references in the garbage collector's roots (*e.g.*, application thread stacks (mutator stacks), static fields, image spaces, zygote space, runtime roots, etc.) are updated with their corresponding to-space addresses. After the STW pause, the garbage collection process then proceeds to the concurrent compaction phase.

In the concurrent compaction phase, the execution of mutators will resume and the garbage collector thread will start processing the to-space pages of the heap one-by-one. If a mutator accesses an area (*e.g.*, a to-space page) in the heap that has not yet been processed by the garbage collector thread, and thus does not have a page allocated, the mutator will receive a SIGBUS signal (bus error) indicating a page fault. Responsive to receiving the page fault, a page buffer (*e.g.*, 4KB page buffer) is created. All the reachable objects that should be located on the missing page, as per the computations done in the concurrent computation phase, are copied to the page buffer and the references inside these objects are updated to the corresponding new addresses. Finally, the user space invokes the *userfaultfd* input/output control (*ioctl*) system call to hand over the page buffer to the kernel, including an indication of the page to make visible on the faulting address. In response, the kernel can copy the contents of the page buffer to a page and map that page. The kernel then resumes execution of the mutator.

In conclusion, utilizing the Linux *userfaultfd* system call in a garbage collection process is an efficient replacement for the revoke access rights process (first compaction approach) and the

page fault process (second compaction approach) described earlier, resulting in reliable garbage collection requiring less overhead while continuing to provide and offering more data consistency (reliability).

References:

[1] Cracauer, Martin. "Generational Garbage Collection, Write Barriers/Write Protection and userfaultfd(2)." Publication Date: August 13, 2016. Retrieved from: https://medium.com/@MartinCracauer/generational-garbage-collection-write-barriers-write-protection-and-userfaultfd-2-8b0e796b8f7f.

[2] Iyengar, Balaji & Gehringer, Edward F., & Wolf, Michael, & Manivannan, Karthikeyan. "Scalable Concurrent and Parallel Mark." ACM SIGPLAN Notices, Vol. 47, Issue 11. Publication Date: June 2012. Retrieved from: https://doi.org/10.1145/2426642.2259006.

[3] Hussein, Ahmed & Payer, Mathias & Hosking, Antony L. & Vick, Chris. "One Process to Reap Them All: Garbage Collection as-a-Service." Publication Date: April 7-8, 2017. Retrieved from: https://doi.org/10.1145/3050748.3050754.

[4] Kermany, Haim & Petrank, Erez. "The Compressor: concurrent, incremental, and parallel compaction." PLDI '06: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. Publication Date: June 11-14, 2006. Retriepved from: https://doi.org/10.1145/1133981.1134023.

[5] Österlund, Erik. "Going Beyond On-The-Fly Garbage Collection and Improving Self-Adaptation with Enhanced Interfaces." (PhD dissertation). Linnaeus university press, Växjö. Publication Date: 2019. Retrieved from: http://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-89999.

[6] Patent Publication: US20160098229A1. Automatic analysis of issues concerning automatic memory management. Publication date: October 1, 2014.