Efficient Synchronization of Linux Memory Regions over a Network: A Comparative Study and Implementation

A user-friendly approach to application-agnostic state synchronization

Felicitas Pojtinger (Stuttgart Media University)

2023-08-04

THIS IS A LLM-GENERATED TEXT VERSION OF THE NOTES FOR ESTIMATING THE EXPECTED THESIS LENGTH. THIS IS NOT THE FINISHED DOCUMENT AND DOES NOT CITE SOURCES.

Introduction

Introduction

- Research question: Could memory be the universal way to access and migrate state?
- Why efficient memory synchronization is the missing key component
- High-level use cases for memory synchronization in the industry today

Technology

Page Faults

Page faults are a notable aspect of computer memory management, essentially serving as system triggers when a process attempts to access a section of memory that is not yet mapped into its address space. These events are pivotal within the broader scope of a process' execution lifecycle, providing signals for when a process requires access to a particular memory segment.

The event of a page fault allows the memory management unit (MMU) of an operating system (OS) to fetch the desired memory chunk from a remote location, mapping it to the address where the page fault occurred. This is essentially on-demand fetching of data, ensuring efficiency in memory usage by only retrieving required data.

Typically, page fault handling is a role reserved for the operating system's kernel, the core component of an OS responsible for managing low-level tasks such as memory management. As M. V. Wilkes noted in his seminal

Delta Synchronization

File synchronization is a crucial aspect of managing data across multiple devices and networks. The widespread acceptance and application of a tool such as rsync, arguably the most prevalent in this realm, is testament to the need for efficient methods of synchronizing files. A key feature that makes rsync an invaluable tool is its delta-transfer algorithm, a method that allows for swift synchronization of changes between a local and remote file.

At the heart of the delta sync algorithm is the concept of file block division. It starts with splitting the file on the destination side into fixed-size blocks, thereby setting up the foundation for the comparison between the local and remote files. This process may appear simple, but it provides a powerful basis for achieving file synchronization.

A subsequent step involves calculating a weak, yet swift checksum for each block in the destination file. The calculated checksum is a compact

File Systems In Userspace (FUSE)

File Systems in Userspace (FUSE) is a unique software interface for Unix-like operating systems that lets non-privileged users create and manage their file systems without altering kernel code. The FUSE API makes this possible, fostering a clear separation between the kernel and user space.

To utilize the FUSE API, a userspace program first registers itself with the FUSE kernel module. This program then provides callbacks for numerous filesystem operations such as open, read, and write. In “Design and Implementation of a Ceph Filesystem FUSE Client” (2010), authors Zhu, Chen, and Jiang explain how these callbacks handle the necessary operations on the filesystem, facilitating communication between the user and the kernel module.

When a user performs a filesystem operation on a FUSE-mounted filesystem, the kernel module sends a request for the operation to the

Network Block Device (NBD)

Network Block Device (NBD) is a protocol that facilitates communication between a server and a client, providing access to block devices over a network. Typically, the server component is provided by the user space, while the NBD kernel module serves as the client.

While NBD protocol can run over wide area networks (WAN), it is designed primarily for local area network (LAN) or localhost usage. The protocol operates in two phases: a handshake and transmission. A handshake establishes connection parameters, while the transmission phase is responsible for actual data exchange.

There are two actors in the NBD protocol: one or multiple clients, and a server. The protocol also introduces a virtual concept of an ‘export’, essentially a block device made available by the server. When a client connects to the server, the server sends a greeting message containing the server’s flags. The client responds with its own flags and an export

Virtual Machine Live Migration

Virtual Machine Live Migration is a technology used to transfer running virtual machines, along with their active state and connected devices, from one host to another with minimal downtime. This technology greatly enhances system flexibility and efficiency, offering benefits such as load balancing, hardware maintenance, and fault management.

Pre-Copy

A key strategy in the field of live VM migration is the pre-copy algorithm, an approach which minimizes downtime by duplicating data from the source to the destination while the virtual machine continues to run. As affirmed by Hines and Gopalan in their paper “Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning” (2009), this approach allows for the optimization of chunk fetching beyond what the Network Block Device (NBD) protocol over WAN can offer.

The pre-copy migration process begins with the copying of the initial state of the VM’s memory to the destination. During this operation, if any chunks of data are modified, they are marked as ‘dirty’. These dirty chunks are then copied to the destination, continuing until the number of remaining chunks is reduced sufficiently to satisfy a maximum downtime criterion. At this point, the VM is suspended on the source, and the remaining chunks are synchronized to the destination. Upon completion

Post-Copy

The post-copy migration strategy is a notable alternative to the pre-copy approach in the context of Virtual Machine Live Migration. It operates on a fundamentally different principle, transferring the virtual machine (VM) to the destination with only a minimal set of chunks and then fetching additional data as required.

In the post-copy migration, the VM is initially suspended on the source and then relocated to the destination with a limited set of data chunks. Upon reaching the destination, the VM resumes operation. As explained by Hu et al. in “Performance Evaluation of Virtual Machine Live Migration” (2010), when the VM attempts to access a chunk that is not available at the destination, a page fault occurs. This fault triggers the fetching of the missing page from the source, after which the VM execution continues.

One of the key advantages of post-copy migration is that it eliminates the need to retransmit dirty chunks to the destination before hitting the

Workload Analysis

The paper “Reducing Virtual Machine Live Migration Overhead via Workload Analysis” offers an intriguing exploration of the timing of migration events, and while it predominantly targets virtual machines (VMs), its findings can potentially extend to other applications or migration scenarios.

This paper introduces a method that identifies the workload cycles of VMs and uses this information to decide whether to postpone migration. The method operates by analysing the cyclic patterns that can unnecessarily extend a VM’s migration time. It subsequently identifies optimal cycles in which to conduct the migration.

In the context of VMs, such cycles could arise due to various events, such as the garbage collection (GC) of a large application triggering substantial changes to the VM’s memory. When a migration proposal occurs, the system assesses whether it is within a beneficial cycle to carry out the

Planning

Pull-Based Synchronization With userfaultfd

The approach of pull-based synchronization is quite innovative in dealing with page faults in userspace. Recently, Linux kernel 4.11 introduced a new system called userfaultfd, which elegantly enables this process. The userfaultfd system permits the handling of these page faults in userspace, making it possible to handle memory in an innovative and flexible manner.

In practice, the memory region for handling is allocated, for example, using the mmap system call. This allows the creation of new memory regions for use, which is the first step in creating a userspace page fault handler. Once the memory region is prepared, the file descriptor for the userfaultfd API is obtained. The next crucial step is to transfer this file descriptor to a process that should respond with the memory chunks that are intended to be put into the faulting address. This operation typically involves inter-process communication and synchronization, a challenge that necessitates careful programming and error handling.

Push-Based Synchronization With mmap and Hashing

The notion of push-based synchronization, specifically using mmap and hashing, offers an alternative approach to handle memory synchronization between systems. Rather than responding to page faults as in userfaultfd, this technique employs a file to track modifications to a memory region. By synchronizing this representative file between systems, we achieve memory region synchronization.

This approach mirrors how Linux utilizes swap space, enabling chunks of memory to be moved to disk or another swap partition when high-speed RAM is not a requirement. A process referred to as “paging out”. Moreover, Linux can also load missing memory chunks from the disk. This method operates analogously to how userfaultfd handles page faults, but it circumvents user space, potentially enhancing the speed of execution.

To detect changes in files, the Linux kernel offers the inotify system. This system enables applications to register handlers on file events, like WRITE

Push-Pull Synchronization with FUSE

The exploration of push-pull synchronization with FUSE (Filesystem in Userspace) provides an innovative approach to address the challenges related to efficient virtual memory management. Given the fact that push-based synchronization demands intensive CPU and I/O operations and userfaultfd-go only allows low throughput, a more optimized solution is required.

One potential solution could be to design a custom filesystem in Linux, implemented as a kernel module, that could intercept read/write operations to the mmaped region. This approach could allow a custom backend to respond to these operations, eliminating the need for userfaultfd-go or resource-consuming hashing techniques.

However, this approach is not without potential drawbacks. It requires direct operation in the kernel, which presents significant complexities. As the paper “Building Reliable, High-Performance Communication Systems

Pull-Based Synchronization With NBD

Block Device-based synchronization offers an innovative solution to enhance the effectiveness of Virtual Machine (VM) live migration, as it capitalizes on the unique features of block devices in Linux. Unlike the conventional use of files, a block device can be utilized for mmap (memory map), thereby capturing reads/writes to a single mmaped file.

Block devices in Linux are typically storage devices that allow reading and writing fixed chunks, or blocks, of data. This process could be similar to how a file system can be implemented in a kernel module. However, the implementation of such a solution in kernel space may raise several issues related to security, portability, and developer experience, as outlined in “Taming Hosted Hypervisors with (Mostly) Deprivileged Execution” (Belay et al., 2012).

To address these challenges, the utilization of a Network Block Device (NBD) server emerges as a potential solution. The NBD server can be used