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

Introduction

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.

Memory management in Linux plays a crucial role in ensuring efficient utilization of system resources. The Linux kernel employs various mechanisms, such as virtual memory, to provide each process with its own address space. This enables processes to access memory regions specific to their needs, while maintaining isolation and security.

In recent years, there has been a growing interest in exploring memory as a universal storage API. The concept revolves around using memory as a means to access not only traditional data stored in RAM but also other resources, such as files, network resources, and even remote objects. By treating memory as a universal interface, applications can seamlessly access and manipulate diverse resources using familiar memory-based operations.

Pull-Based Memory Synchronization with userfaultfd

Pull-Based Memory Synchronization with userfaultfd

Page faults occur when a process attempts to access a memory region that has not been mapped into its address space. By monitoring these page faults, we can identify when a process intends to access a specific memory segment. Leveraging this information, we can implement a pull-based approach where we retrieve the required memory chunk from a remote location and map it to the address where the page fault occurred. This strategy ensures that data is fetched only when it is needed, optimizing memory utilization.

Typically, the handling of page faults is a responsibility of the kernel. However, in our case, we aim to handle page faults in userspace. Previously, this was achievable by handling the SIGSEGV signal within the process. However, we can now utilize a more elegant solution called userfaultfd, which became available since kernel version 4.11. The userfaultfd system allows for the handling of these page faults in userspace, offering improved control and flexibility.

Push-Based Memory Synchronization with `mmap` and Hashing

Push-Based Memory Synchronization with mmap and Hashing

To address the limitations of the userfaultfd-based memory synchronization approach, an alternative push-based synchronization method can be employed. This approach aims to improve upon userfaultfd by utilizing a file to track changes to a memory region and synchronizing this file between two systems to effectively synchronize the corresponding memory region.

In Linux, the concept of swap space allows the operating system to move chunks of memory to disk or other swap partitions when the fast access of RAM is not required. This process, known as “paging out,” is similar to how the proposed synchronization method works. Linux can load missing chunks from a disk, eliminating the need to go through userspace as required by userfaultfd. Consequently, this approach can potentially offer faster synchronization.

The mmap function in Linux enables the mapping of a file into memory,

Push-Pull Memory Synchronization with a FUSE

To overcome the limitations of the push-based method and the low throughput of userfaultfd-go, an alternative solution is needed. What if it were possible to capture read and write events without relying on userfaultfd-go or performing hashing?

One approach could involve creating a custom file system in Linux and loading it as a kernel module. This file system would intercept reads and writes to and from the mmaped region, allowing custom backends to respond to these events. However, implementing such a system directly in the kernel introduces several potential drawbacks.

Writing kernel modules, traditionally done in C, has its challenges. Although it is now possible to write kernel modules in Rust, many issues persist. Kernel modules lack portability as they are built for specific kernels, making distribution to users difficult. Furthermore, running a file system in the kernel eliminates the need for user space involvement,

Pull-Based Memory Synchronization with NBD

Pull-Based Memory Synchronization with NBD

To address the need for a more efficient API for catching reads and writes to a single mmaped file, an alternative approach can be considered that does not require implementing a complete file system. Instead, a block device can be utilized.

In Linux, block devices are typically storage devices capable of reading and writing fixed chunks of data known as blocks. Similar to mmaping a file, it is possible to mmap a block device. However, block devices are typically implemented as kernel modules or reside in kernel space, introducing similar challenges regarding security, portability, and developer experience as with file system implementations.

Instead of using FUSE, an alternative solution involves creating a Network Block Device (NBD) server. The NBD server can be used by the kernel NBD module, similar to how the process connected to the FUSE kernel module. The key distinction is that the NBD server does not provide a file system; it

Push-Pull Memory Synchronization with Mounts

Push-Pull Memory Synchronization with Mounts

To overcome the challenges of latency when using NBD over a public network, a layer of indirection called a Mount is introduced. The Mount consists of both a client and a server, both running on the local system. This approach allows for smarter pull/push strategies and provides better performance in WAN scenarios.

The server and client are connected using a connected UNIX socket pair, providing a reliable and efficient communication channel between them. By building on this basic direct mount, additional functionalities like file and slice mounts can be implemented, enabling easy usage and integration with sync and msync operations.

These managed mounts provide a higher level of abstraction and flexibility, allowing for more sophisticated synchronization strategies and optimizing data transfer between the client and the remote server.

The use of the mmap/slice approach brings several benefits. Firstly, it

Pull-Based Memory Synchronization with Migrations

Pull-Based Memory Synchronization with Migrations

The managed mounts API we have implemented provides efficient access to a remote resource through memory. However, it is not well suited for migration scenarios where minimizing the maximum acceptable downtime is crucial.

To optimize migration, we need to split the process into two distinct phases. We can still leverage preemptive background pulls and parallelized device/syncer startup, but the push process is dropped. This approach allows us to pull the majority of the data first and then finalize the move later with the remaining data.

This approach is inspired by the pre-copy approach to VM live migration, but also incorporates some benefits of the post-copy approach. One of the challenges we face is concurrent access to the resource by multiple readers or writers, which the mount-based API does not allow safely. This constraint poses a problem for migration because suspending the VM or

Optimizing Mounts and Migrations

Optimizing Mounts and Migrations

Indeed, r3map offers unique advantages compared to existing remote mount and migration solutions. Its transport-agnostic design allows it to be flexible and adaptable to different deployment scenarios.

In LAN environments, where assumptions can be made about security and the absence of bad actors within the subnet, r3map can leverage fast and latency-sensitive protocols like SCSI RDMA (SRP) or custom protocols. This enables efficient and high-performance migration or mount scenarios within a local network.

On the other hand, for WAN deployments where security is a greater concern, r3map can utilize standard internet protocols such as TLS over TCP or QUIC. These protocols provide encryption and secure communication, allowing for migration over the public internet with appropriate security measures in place.

Additionally, r3map's transport-agnostic approach opens up possibilities

Case Studies

Case Studies

ram-dl is a tech demo and experiment built on top of the r3map framework. It utilizes the fRPC backend to expand local system memory by allowing the mounting of a remote system's RAM locally. The primary purpose of ram-dl is to enable the inspection of a remote system's memory contents.

The implementation of ram-dl is based on the direct mount API provided by r3map. It leverages utilities like mkswap, swapon, and swapoff to enable paging out to the block device exposed by the direct mount API.

ram-ul, on the other hand, is a companion tool that facilitates "uploading" RAM by exposing memory, file, or directory-backed files over fRPC. Together, ram-ul and ram-dl demonstrate how r3map can be used for unique use cases, such as accessing real, remote RAM or Swap.

Although ram-dl and ram-ul are not intended for real-world use cases, they serve as demonstrations of the capabilities of r3map. The projects

Conclusion

Conclusion

In conclusion, we have explored various synchronization options and compared their ease of implementation, CPU load, and network traffic. We have discussed the limitations of traditional approaches, such as file synchronization, database synchronization, and custom sync protocols. These methods often require complex sync protocols, suffer from high network traffic, and lack universality.

We then introduced r3map, a novel solution that leverages memory as a universal access format for synchronization. By utilizing the managed mount API, r3map allows for efficient and seamless synchronization of state across different systems. The ability to mount remote memory regions, migrate state between hosts, and stream data on-demand opens up new possibilities for efficient synchronization and access to resources.

This approach offers several advantages, including reduced network traffic, lower CPU load, and the ability to handle various types of state with a