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

TODO: Add subtitle

Felicitas Pojtinger (Stuttgart Media University)

2023-08-04

Introduction

Introduction

TODO: Add introduction

Technology

The Linux Kernel

The Linux kernel, an open-source kernel, was originally brought into existence by Linus Torvalds in 1991. Developed primarily in the C programming language, it has recently seen the addition of Rust as an approved language for further expansion and development[1]. The kernel's influence is extensive, powering millions of devices across the globe, including servers, desktop computers, mobile phones, and a myriad of embedded devices.

In essence, the Linux kernel serves as an intermediary between hardware and applications. Its role is to offer an abstraction layer that simplifies the interaction between these two entities. Moreover, it is engineered for compatibility with a wide array of architectures, such as ARM, x86, RISC-V, and others, which increases its versatility and broadens its reach.

The Linux kernel, though foundational, does not function as a standalone operating system. This role is fulfilled by distributions, which build upon

Linux Kernel Modules

At the heart of the Linux system resides its kernel, an extensible, monolithic component that allows for the integration of kernel modules[2]. Kernel modules are small pieces of kernel-level code that can be dynamically incorporated into the kernel, presenting the advantage of extending kernel functionality without necessitating system reboots.

The dynamism of these modules comes from their ability to be loaded and unloaded into the running kernel as per user needs. This functionality aids in keeping the kernel size both manageable and maintainable, thereby promoting efficiency. Kernel modules are developed using the C programming language, like the kernel itself, ensuring compatibility and consistent performance.

Kernel modules interact with the kernel via APIs (Application Programming Interfaces), mechanisms that facilitate communication between software components. Despite their utility, kernel modules do carry a potential risk.

UNIX Signals and Handlers

UNIX signals are an integral component of UNIX-based systems, including Linux. They function as software interrupts, notifying a process of significant occurrences, such as exceptions. Signals may be generated from various sources, including the kernel, user input, or other processes, making them a versatile tool for inter-process communication.

Despite their notification role, signals also serve as an asynchronous communication mechanism between processes or between the kernel and a process. As such, they have an inherent ability to deliver important notifications without requiring the recipient process to be in a specific state of readiness[4]. Each signal has a default action associated with it, the most common of which are terminating the process or simply ignoring the signal.

To customize how a process should react upon receiving a specific signal, handlers can be utilized. Handlers are routines that dictate the course of

Memory Hierarchy

The memory hierarchy in computers is an organized structure based on factors such as size, speed, cost, and proximity to the Central Processing Unit (CPU). It follows the principle of locality, which suggests that data and instructions that are accessed frequently should be stored as close to the CPU as possible[6]. This principle is crucial primarily due to the limitations of “the speed of the cable”, where both throughput and latency decrease as distance increases due to factors like dampening and the finite speed of light.

At the top of the hierarchy are registers, which are closest to the CPU. They offer very high speed, but provide limited storage space, typically accommodating 32-64 bits of data. These registers are used by the CPU to perform operations.

Following registers in the hierarchy is cache memory, typically divided into L1, L2, and L3 levels. As the level increases, each layer becomes larger and

Memory Management in Linux

Memory management forms a cornerstone of any operating system, serving as a critical buffer between applications and physical memory. Arguably, it can be considered one of the fundamental purposes of an operating system itself. This system helps maintain system stability and provides security guarantees, such as ensuring that only a specific process can access its allocated memory.

Within the context of the Linux operating system, memory management is divided into two major segments: kernel space and user space.

Kernel space is where the kernel, kernel extensions, and device drivers operate. The kernel memory module is responsible for managing this segment. Slab allocation is a technique employed in kernel space management. This technique groups objects of the same size into caches, enhancing memory allocation speed and reducing fragmentation of memory[8].

Swap Space

Swap space refers to a designated portion of the secondary storage utilized as virtual memory in a computer system[9]. This feature plays a crucial role in systems that run multiple applications simultaneously. When memory resources are strained, swap space comes into play, relocating inactive parts of the RAM to secondary storage. This action frees up space in primary memory for other processes, enabling smoother operation.

In the Linux operating system, swap space implementation aligns with a demand paging system. This means that memory is allocated only when required. The swap space in Linux can be a swap partition, which is a distinct area within the secondary storage, or it can take the form of a swap file, which is a standard file that can be expanded or truncated based on need. The usage of swap partitions and files is transparent to the user.

The Linux kernel employs a Least Recently Used (LRU) algorithm to

Page Faults

Page faults are instances in which a process attempts to access a page that is not currently available in primary memory. This situation triggers the operating system to swap the necessary page from secondary storage into primary memory[10]. These are significant events in memory management, as they determine how efficiently an operating system utilizes its resources.

Page faults can be broadly categorized into two types: minor and major. Minor page faults occur when the desired page resides in memory but isn't linked to the process that requires it. On the other hand, a major page fault takes place when the page has to be loaded from secondary storage, a process that typically takes more time and resources.

To minimize the occurrence of page faults, memory management algorithms such as Least Recently Used (LRU) and the more straightforward clock algorithm are often employed. These algorithms

mmap

The mmap is a versatile UNIX system call, used for mapping files or devices into memory, enabling a variety of core tasks like shared memory, file I/O, and fine-grained memory allocation. Due to its powerful nature, it is commonly harnessed in applications like databases. It is indeed a “power tool”, mandating the careful and intentional use to avoid issues or inefficiencies.

One standout feature of mmap is its ability to function by creating a direct memory mapping between a file and a region of memory[12]. This connection means that read operations performed on the mapped memory region directly correspond to reading the file and vice versa, enhancing efficiency by reducing the overhead as the necessity for context switches diminishes.

A key advantage that mmap provides is the capacity to facilitate zero-copy operations. In practical terms, this signifies data can be accessed directly

inotify

The `inotify` is an event-driven notification system of the Linux kernel, designed to monitor the file system for a multitude of events, such as modifications and accesses, among others[13]. It's unique for its intrinsic ability to heed to specific events through a watch feature. For instance, it can be configured to watch only write operations on certain files. This level of control can offer considerable benefits in cases where there is a need to focus system resources on certain file system events, and not on others.

Naturally, `inotify` comes with some recognizable advantages. Significantly, it diminishes overhead and resource use when compared to polling strategies. Polling is an operation-heavy approach as it continuously checks the status of the file system, regardless of whether any changes have occurred. In contrast, `inotify` works in a more event-driven way, where it only takes action when a specific event occurs. This differential action is inherently more efficient, reducing overhead where there are

Linux Kernel Disk and File Caching

Linux Kernel Disk and File Caching are key elements of the Linux operating system that work to boost efficiency and performance. Within this framework, there are two broad categories: disk caching and file caching.

Disk caching in Linux is a strategic method that temporarily stores frequently accessed data in Random Access Memory (RAM). It is implemented through the page cache subsystem in Linux. This approach leverages the principle of locality, which states that odds are good that data situated near data that has already been accessed will be needed soon. This method of retaining data close to the CPU where it may be swiftly accessed without costly disk reads greatly reduces overall access time. The data within the cache is managed using the Least Recently Used (LRU) algorithm, which prunes the least recently used items first when space is needed.

File caching, on the other hand, is where Linux caches file system

TCP, UDP and QUIC

TCP (Transmission Control Protocol), UDP (User Datagram Protocol), and QUIC (Quick UDP Internet Connections) are key communication protocols utilized in network communications, each showcasing distinct characteristics and use cases.

TCP has long been the reliable backbone for internet communication due to its connection-oriented nature [14]. This protocol ensures the guaranteed delivery of data packets and their correct order, rendering it a highly dependable means for data transmission. Significantly, TCP incorporates error checking, allowing the detection and subsequent retransmission of lost packets. Moreover, TCP includes a congestion control mechanism to manage data transmission seamlessly during high traffic. Owing to these features, TCP is widely used to power the majority of the web where reliable, ordered, and error-checked data transmission is paramount.

Delta Synchronization

Delta synchronization is an innovative technique that allows for efficient synchronization of files between hosts, aiming to transfer only those parts of the file that have undergone changes instead of the entire file. This incremental approach can remarkably reduce network and I/O overhead, optimally utilizing system resources.

Perhaps the most recognized tool employing this method of synchronization is rsync, an open-source data synchronization utility in Unix-like operating systems. The elixir offered by rsync lies in its usage of a delta-transfer algorithm, which astutely computes the difference between local and remote files, thereafter synchronizing the changes.

Fundamentally, the algorithm embarks on file block division, dissecting the file on the destination side into fixed-size blocks. For each of these blocks, a quick albeit weak checksum calculation is performed, and these checksums are transferred to the source system.

File Systems In Userspace (FUSE)

File Systems in Userspace (FUSE) is a software interface that enables the creation of custom file systems in the userspace, as opposed to developing them as kernel modules. It liberates developers from the obligatory low-level kernel development normally associated with creating new file systems.

FUSE is available on various platforms. Though mostly deployed on Linux, it can also be found on macOS and FreeBSD. The secret to developing file systems in userspace lies in the FUSE Application Programming Interface (API). In this setup, a userspace program aligns itself with the FUSE kernel module and provides callbacks for the file system operations. These operations can include tasks like open, read, write, and many others.

So, when a user performs a file system operation on a mounted FUSE file system, the kernel module sends a request for executing that operation to the userspace program. This is followed by the userspace program

Network Block Device (NBD)

Network Block Device (NBD) embodies a protocol that enables communication between a user space-provided server and a Kernel-provided client. Though potentially deployable over Wide Area Networks (WAN), it is primarily designed for Local Area Networks (LAN) or localhost usage. The protocol is divided into two phases: the handshake and the transmission[19].

The execution of the NBD protocol involves multiple participants, notably one or several clients, a server, and the virtual concept of an export. It initiates with a client establishing a connection with the server. The server reciprocates by delivering a greeting message highlighting various server flags. The client responds by transmitting its own flags along with the name of an export to use; a single NBD server can expose multiple devices.

Upon receiving this, the server sends the size of the export and other metadata. The client acknowledges this data, completing the handshake.

Pre-Copy

Virtual Machine Live Migration has emerged as a key strategy to optimize systems involving WAN, with a focus on minimizing downtime during data transfer processes. It essentially involves the shifting of a virtual machine, its state, and its connected devices from one host to another, with the objective to minimize disrupted service.

Live migration algorithms can be categorized into two broad types: pre-copy migration and post-copy migration. Here we delve into the specifics of the pre-copy migration technique.

The foremost characteristic of pre-copy migration is its ‘run-while-copy’ nature. In other words, the copying of data from the source to the destination occurs concurrently while the virtual machine (VM) continues to operate. This method is also applicable in a generic migration context where an application or another data state is being updated.

In the case of a VM, the pre-copy migration procedure commences with

Post-Copy

Post-copy migration is an alternative approach to pre-copy migration in Virtual Machine Live Migration. While pre-copy migration operates by copying data before the VM halt, post-copy migration opts for another strategy: it immediately suspends the VM operation on the source, moves it to the destination, and resumes it – all with only a minimal subset of data or ‘chunks’.

The process initiates with the prompt suspension of the VM on the source system. It is then relocated to the destination with only some essential chunks of data involved in the transfer. Upon reaching the destination, the VM operation is resumed.

During this resumed operation, whenever the VM attempts to access a chunk of data not initially transferred during the move, a page fault arises. A page fault, in this context, is a type of interrupt generated when the VM tries to read or write a chunk that is not currently present in its address

Workload Analysis

The research article “Reducing Virtual Machine Live Migration Overhead via Workload Analysis” explores strategies to determine the most suitable timing for virtual machine (VM) migration[21]. Even though the study chiefly focuses on virtual machines, the methodologies proposed could be adapted for use with various other applications or migration circumstances.

The method proposed in the study identifies cyclical workload patterns of VMs and leverages this knowledge to delay migration when beneficial. This is achieved by analyzing recurring patterns that may unnecessarily postpone VM migration, and identifying optimal cycles within which VMs can be migrated. For instance, in the realm of VM usage, such cycles could be triggered by a large application’s Garbage Collection (GC) that results in numerous changes to VM memory.

When migration is proposed, the system verifies whether it is in an

Streams and Pipelines

Streams and pipelines are fundamental constructs in computer science, enabling efficient, sequential processing of large datasets without the need for loading the entire dataset into memory. They form the backbone of modular and efficient data processing techniques, each concept having its unique characteristics and use cases.

A stream represents a continuous sequence of data, serving as a conduit between different points in a system. Streams can be either a source or a destination for data. Examples include files, network connections, and standard input/output devices, amongst others. The power of streams comes from their ability to process data as it becomes available. This aspect allows for minimization of memory consumption, making streams particularly impactful for scenarios involving long-running processes where data is streamed over extended periods of time[22].

On the other hand, a pipeline comprises a series of data processing

gRPC

The gRPC is an open-source, high-performance remote procedure call (RPC) framework developed by Google in 2015. The gRPC framework is central to numerous data communication processes and is recognized for its cross-platform compatibility, supporting a variety of languages including Go, Rust, JavaScript and more. The gRPC maintains its development under the auspices of the Cloud Native Computing Foundation (CNCF), further emphasizing its integration within the realm of cloud services.

One of the notable features of the gRPC is its usage of HTTP/2 as the transport protocol. This allows it to exploit beneficial capabilities of HTTP/2 such as header compression, which minimizes bandwidth usage, and request multiplexing, enabling multiple requests to be sent concurrently over a single TCP connection.

In addition to HTTP/2, gRPC utilizes Protocol Buffers (protobuf) as the

Redis

Redis, an acronym for ‘REmote DIctionary Server,’ is an in-memory data structure store, primarily utilized as a database, cache, or message broker. Introduced by Salvatore Sanfilippo in 2009, Redis lends itself to high-speed and efficient data management, thus positioning itself as a key resource in numerous applications demanding rapid data processing.

Setting it apart from other NoSQL databases, Redis supports a multitude of data structures, including lists, sets, hashes, and bitmaps. These differing data types allow for versatile data manipulation capabilities. Thus, in scenarios where a diverse set of data types need to be managed concurrently, Redis supplies a fitting solution[25].

One of the primary drivers for Redis’s speed is its reliance on in-memory data storage. By storing data in memory rather than on disk, Redis enables low-latency reads and writes, contributing to the technology’s overall capacity for swift, efficient data operations.

S3 and Minio

Amazon’s Simple Storage Service (S3) and Minio are two contemporary solutions providing storage functionality, each with unique characteristics and use cases.

S3 is a scalable object storage service crafted for data-intensive tasks, befitting anything from small deployments to large-scale applications. It is one of the prominent services offered by Amazon Web Services, a leading provider of cloud computing resources. S3’s design allows for global distribution, which means the data can be stored across multiple geographically diverse servers. This permits fast access times from virtually any location on the globe, crucial for globally distributed services or applications with users spread across different continents.

S3 offers a variety of storage classes catering to different needs, whether the requirement is for frequent data access, infrequent data retrieval, or long-term archival. This flexibility ensures the service can meet a wide

Cassandra and ScylllaDB

Apache Cassandra and ScyllaDB are compelling wide-column NoSQL databases, each tailored for large-scale, distributed data management tasks. They blend Amazon's Dynamo model's distributed nature with Google's Bigtable model's structure, leading to a highly available, heavy-duty database system.

Apache Cassandra is lauded for its scalability and eventual consistency, designed to handle vast amounts of data spread across numerous servers. Unique to Cassandra is the absence of a single point of failure, thus ensuring continuous availability and robustness—critical for systems requiring high uptime.

Cassandra's consistency model is tunable according to needs, ranging from eventual to strong consistency. It distinguishes itself by not employing master nodes due to its usage of a peer-to-peer protocol and a distributed hash ring design. These design choices eradicate the