

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

A user-friendly approach to application-agnostic state synchronization

Felicitas Pojtinger (Stuttgart Media University)

2023-08-04

Unsorted Research Questions

Structure

- Introduction
 - Memory management in Linux
 - Memory as the universal storage API
 - What would be possible if memory would be the universal way to access resources?
 - Why efficient memory synchronization is the missing key component
 - High-level use cases for memory synchronization in the industry today
- Pull-Based Memory Synchronization with userfaultfd
 - Plain language description of userfaultfd (what are page faults)
 - Exploring an alternative method by handling page faults using signals
 - Handlers and registration
 - History of userfaultfd
 - Allocating the shared region
 - Maximum shared region size is limited by available physical memory
 - Transferring handler sockets between processes
 - Implementing userfaultfd bindings in Go

Content

- Pull-Based Memory Synchronization with `userfaultfd`
 - Page faults occur when a process tries to access a memory region that has not yet been mapped into a process' address space
 - By listening to these page faults, we know when a process wants to access a specific piece of memory
 - We can use this to then pull the chunk of memory from a remote, map it to the address on which the page fault occurred, thus only fetching data when it is required
 - Usually, handling page faults is something that the kernel does
 - In our case, we want to handle page faults in userspace
 - In the past, this used to be possible by handling the `SIGSEGV` signal in the process
 - In our case however, we can use a recent system called `userfaultfd` to do this in a more elegant way (available since kernel 4.11)
 - `userfaultfd` allows handling these page faults in userspace
 - Implementing this in Go was quite tricky, and it involves using `unsafe`
 - We can use the `syscall` and `unix` packages to interact with `ioctl` etc