Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

System Calls for Containerising and Managing Processes in Linux

A Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science (M.Sc.) in Applied Computer Science

at the

Berlin University of Applied Sciences (HTW)

First Supervisor:	Prof. Dr. Sebastian Bauer
Second Supervisor:	Dr. Michael Witt
Author:	B.Sc. Atanas Denkov
Matriculation Number:	s0559025
Submission Date:	12.07.2022

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 1
1.3 Content Structure 1

2 Fundamentals **2**
2.1 Virtualisation 2
2.1.1 Axioms 2

List of Figures **I**

List of Tables **II**

Source Code Content **III**

References **IV**

Chapter 1

Introduction

1.1 Motivation

Primitive support for multiprocessing in the form of basic context switching and dedicated I/O components was introduced in the late 1950s. Multiprocessing allowed for concurrent execution of multiple instructions at the cost of increased system complexity. Interleaved processes had a global unrestricted view of the system which inevitably led to unpredictable program behaviour. For example, programs had the ability to modify each other's memory and monopolise computer resources. Hence, to ensure correctness, every program had to carefully manage its interactions with hardware and all other processes in the system, which resulted in an unsustainable programming model.
The aforementioned issues were addressed by shifting the responsibility of resource management and process protection into a privileged control program that acted as an intermediary between hardware and user programs.

1.2 Objectives

1.3 Content Structure

Chapter 2

Fundamentals

2.1 Virtualisation

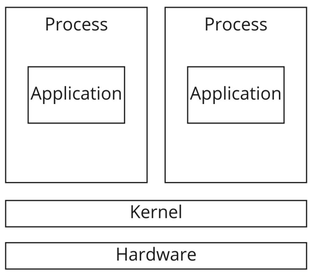
2.1.1 Axioms

Isolation

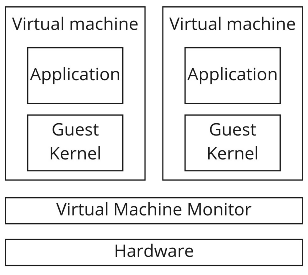
Codd et al. [Cod59] summarise the fundamental requirements of a functional control program and emphasise the concept of noninterference between processes across space and time. Spatial and temporal noninterference can be seen as different qualitative measures of the control program's effectiveness to keep processes safe. Whereas the former deals with the mechanisms that protect references to memory, disk and I/O devices, the latter deals with the allocation of execution time and the protection against the monopolisation thereof.

The STRETCH system [Cod59], albeit quite old, employs an architecture used by modern kernels to guarantee noninterference. The author describes an interruption system that can transfer execution to a different memory address whenever a condition of the machine or process changes, e.g an I/O device emits a signal or the process attempts divison by zero, respectively. The address holds the start instruction of a privileged routine that can react to the changed condition. For example, the routine could serialise access to an I/O device, decode the bit stream and copy it into a memory block local to the process that issued the request. Serialising access ensures that concurrently executing processes cannot spatially interfere with the request. Scheduling the next request to be processed so that all programs make equal or similar runtime progress guarantees temporal noninterference. It is important to note that, by definition, the kernel is considered trustworthy and is allowed to access and modify space assigned to user processes. In other words, noninterference between the kernel and user processes is not assured. Therefore, if the kernel is compromised or encounters an unrecoverable error condition, all user processes become untrustworthy or unavailable, respectively.

Popek and Goldberg [PG74] refer to the control program as a virtual machine monitor that ensures noninterference by providing every program with an environment that is "[...] effect identical with that demonstrated if the program had been run on the original machine directly" [PG74, p. 2]. This definition implies that a running program does not directly use the bare metal resources available. Instead, resources are emulated by the virtual machine monitor at the instruction level and presented as a dedicated physical



(a) Virtualisation using a shared kernel that constantly assures noninterference by handling service requests from user processes. Kernel has full control of the system and represents a single point of failure. There are no mechanisms to reestablsh system health.



(b) Virtualisation using multiple independent kernels, each ensuring noninterference. If one kernel fails, the virtual machine monitor can reclaim the respective resources or perform operations that reconstruct the virtual machine's state from a health checkpoint.

Figure 2.1: Architectural comparison between shared-kernel virtualisation (2.1a) and multi-kernel virtualisation using a virtual machine monitor (2.1b)

system. Such an environment is called a *virtual machine*. Notice that a virtual machine is capable of hosting a kernel, referred to as a *guest kernel*, similar to the one described in the previous paragraph. Consequently, the isolation boundary between user programs running on different virtual machines is stronger compared to user processes running on a shared kernel. Even if a guest kernel fails, other virtual machines remain unaffected. Popek and Goldberg [PG74] define a requirement that the instruction-set architecture of a computer has to satisfy for it to be virtualisable. The instruction set must be segregated into three groups of instructions - privileged, sensitive and innocuous. An instruction is privileged if it requires changing the mode of execution from user to supervisor mode by means of a trap. An instruction i is control-sensitive if, when applied to the current processor state S_1, results in a new state $i(S_1) = S_2$ such that the execution mode of S_2 does not equal that of S_1 or if S_2 has access to different resources than S_1 or both [PG74]. An instruction is behaviour-sensitive if its execution depends on the execution mode or its position in memory. An instruction is innocuous if it is not sensitive. Given these definitions, a computer is virtualisable "[...] if the set of sensitive instructions for that computer is a subset of the set of privileged instructions" [PG74, p. 6]. If this criterion is met, the virtual machine monitor can trap all sensitive instructions and emulate each via a homomorphism $i : C_r \to C_v$ that maps the state space of the processor without the virtual machine monitor loaded C_r to the state space with the virtual machine monitor loaded C_v. Innocuous instructions do not require protection, i.e a homomorphic mapping, and are directly executed by the processor.

Asd asd asd

Performance

List of Figures

2.1 Architectural comparison between shared-kernel virtualisation (2.1a) and multi-kernel virtualisation using a virtual machine monitor (2.1b) 3

List of Tables

Source Code Content

References

[Cod59] E. F. Codd et al. “Multiprogramming STRETCH: Feasibility Considerations”. In: *Commun. ACM* 2.11 (Nov. 1959), pp. 13–17. ISSN: 0001-0782. DOI: 10.1145/368481.368502. URL: https://doi.org/10.1145/368481.368502.

[PG74] Gerald J. Popek and Robert P. Goldberg. “Formal Requirements for Virtualizable Third Generation Architectures”. In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: https://doi.org/10.1145/361011.361073.