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:	04.10.2022

Contents

Chapter 1

Introduction

1.1 Motivation

Primitive support for multiprocessing in the form of basic context switching and dedicated input-output 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

Chapter 2.1.1 introduces the fundamental axioms that define virtualisation - noninterference, isolation and performance. Chapter 2.1.2 presents hardware virtualisation as a concept, discusses the mechanisms it uses to satisfy the axioms, and highlights the trade-offs intrinsic to its design. Chapter 2.1.3 describes a more coarse-grained approach - operating system virtualisation - that has become the de-facto way of managing cloud workloads in multitenant environments.

2.1.1 Axioms

Noninterference

Codd et al. [Cod59] summarise the fundamental requirements of a multiprogramming system and emphasise the concept of noninterference between processes across space and time. *Spatial noninterference* is represented by all mechanisms that protect references to memory, disk and input-output devices [Cod59]. For example, memory segmentation is a method found in operating system kernels that assigns each process a dedicated portion of physical memory that is invisible to all other processes in the system. The kernel traps any attempt made by a process to access memory outside its allocated memory segment, thereby guaranteeing spatial noninterference [SGG18]. *Temporal noninterference* refers to those mechanisms that allocate execution time and protect against the monopolisation thereof [Cod59]. For instance, CPU scheduling is a technique that decides which process shall run on a core such that the core does not idle and all processes make sufficient runtime progress [SGG18]. The scheduling semantics, paired with an interrupt mechanism that makes sure that no process has hold of the core for too long, guarantee temporal noninterference.

Isolation

Anjali, Caraza-Harter, and Swift [ACS20] define isolation as the level of dependency that a virtualisation platform has towards the host kernel. We generalise this definition and say that *isolation* is the level of dependency that one piece of software has to another. Conceptually, isolation deals with explicit vertical relationships between software, and noninterference deals with implicit horizontal relationships between processes. Isolation can be quantified by counting the lines of external source code that a software executes to obtain a particular functionality. For example, Anjali, Caraza-Harter, and Swift [ACS20] count the lines of kernel code that a virtualisation platform executes when providing services to sandboxed applications. High counts indicate a strong dependency, i.e weak isolation, towards the kernel.

Performance

Randal [Ran20] defines performance as the contention between the overhead associated with isolating a process from its environment and the benefits of sharing resources between processes, i.e fully utilising the capacity of the underlying resource pool. Anjali, Caraza-Harter, and Swift [ACS20] use the same definition and contrast the isolation mechanisms provided by three different virtualisation platforms against processing unit, memory and input-output performance metrics. In particular, the authors define an application that computes prime numbers up to a limit. Since the workload is compute-bound, processing speed is measured and compared to the number of executed lines of code that reside in the `/arch`, `/virt` and `/arch/x86/kvm` subsystems of the Linux kernel. Manco et al. [Man17] use *same-host density* as a performance metric that measures the number of sandboxed applications that can be consolidated onto a single server. In addition, *boot, pause and unpause times* are also considered to be important performance indicators for particular use cases, such as elastic content delivery networks [Kue17] [Man17] and serverless computing.

2.1.2 Hardware Virtualisation

Popek and Goldberg [PG74] refer to the control program as a *virtual machine monitor* that ensures isolation and 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 hardware level and presented as a dedicated physical system. Such an environment is called a *virtual machine*.

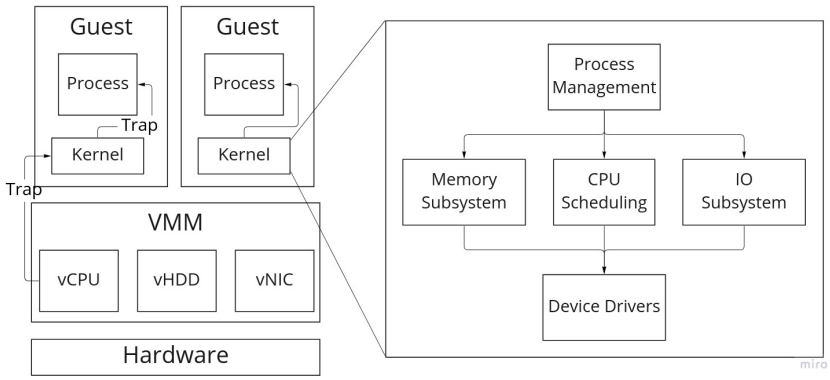

Figure 2.1: Hardware virtualisation architecture. Each guest runs a complete operating system. Privileged operations are trapped by the virtual machine monitor and emulated to provide hardware services.

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 [PG74]. 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 [PG74]. 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 [PG74]. Innocuous instructions do not require protection, i.e a homomorphic mapping, and are directly executed by the processor [PG74].

Given the aforementioned homomorphism, a virtual machine can host a *guest kernel* (Figure 2.1) that runs completely in user mode. Whenever the guest kernel attempts to execute a privileged instruction, the virtual machine monitor traps the attempt and emulates the instruction. Consequently, the guest kernel does not have to be a part of the trusted computing base. Even if it is compromised or encounters an unrecoverable error condition, other virtual machines remain unaffected. As a result, the isolation boundary between user programs running in different virtual machines is stronger compared to processes running on a shared kernel.

In order to fully guarantee spatial noninterference between processes, the virtual machine monitor must be in full control of the host system's memory. There are two primary methods to do this - *shadow paging* and *extended page tables*. The former mechanism is considered first. The virtual

machine monitor maintains a nested page table per guest, also called a *shadow page table* [SN05]. In turn, the guest kernel maintains a page table per process. Whenever the guest kernel schedules a new process for execution, it modifies the *page-table base register* to point to the page table for that process [SN05]. The virtual machine monitor intercepts this attempt and transparently updates the page table pointer to point to the guest's shadow page table corresponding to that process [SN05]. Note that the virtual machine monitor has to traverse the shadow page table for that guest in order to find the nested entry corresponding to the process. Afterwards, the memory management unit takes care of translating the virtual memory addresses of the guest and updating the *translation lookaside buffer*. Alternatively, the memory management unit may be "virtualisation-aware" in the sense that it knows there are two page tables it needs to traverse - the page table that maps guest virtual memory to guest "physical memory", and the page table that maps guest physical memory to actual physical memory. The former is maintained by the guest kernel, whilst the latter is maintained by the virtual machine monitor. The extended page table approach is up to 50% faster than shadow paging [Esx06] because table walks are done in hardware - by the memory management unit. Nevertheless, maintaining page table data structures inside the virtual machine monitor and the guests leads to memory pressure, which is further amplified by the fact that guests, their applications and the virtual machine monitor all share the same physical memory [SGG18].

The spatial noninterference property necessitates that the virtual machine monitor manage all input-output devices and their interactions with the guests. This is accomplished by the already introduced trap-and-emulate pattern. When an application within a virtual machine issues a system call requesting some form of input-output, the request is processed by the I/O stack inside the guest. At the lowest level of the stack, the device driver issues a command to the device, typically by writing to memory specifically assigned to the device, or by calling specific input-output instructions [SGG18]. Either way, the virtual machine monitor intercepts this and traverses its own I/O stack, which remaps guest and real input-output addresses and forwards the request to a physical device [WR12]. After processing the request, the physical device triggers an interrupt that is caught by the virtual machine monitor and transformed into a virtual equivalent that is sent to the virtual machine that issued the request. To reduce the overhead associated with interrupt processing, the virtual machine monitor can batch multiple events together and use a single interrupt to notify the guest kernel [WR12]. Still, a request must traverse two input-output stacks. The same holds for the response. In addition, hardware optimisations such as direct memory access are emulated in software, which further degrades performance. This, however, can be mitigated by integrating an input-output memory management unit that remaps all direct memory accesses of a device on the host to an address space in the guest.

The cost of hardware virtualisation becomes apparent when measuring same-host density and boot times. Manco et al. [Man17] consider memory consumption and on-disk image size as the primary limiting factors. The authors measure the time it takes to create and boot virtual machines using the Xen virtual machine monitor and show the negative effects that on-disk

image size has by starting images with varying sizes by manually "[...] injecting binary objects into the uncompressed image file" [Man17, p. 3]. As the number of consolidated virtual instances increases and the image size grows, creation and boot times increase linearly. Furthermore, the authors show that creating and starting a process directly on the host is, on average, two orders of magnitude faster. Lv et al. [Lv12] also evaluate Xen and state that processing units spend 25% of their total cycles in hypervisor mode instead of executing guest applications when running "[...] SPEC's first benchmark addressing performance evaluation of datacenter servers used in virtualised server consolidation" [Lv12, p. 2], which includes components such as a web, database and application server.

2.1.3 Operating System Virtualisation

Operating system virtualisation refers to all mechanisms that enable the creation of secure and isolated application environments that run on top of a shared kernel. Conventionally, these mechanisms are baked into the kernel and are therefore part of the trusted computing base. The kernel may expose these through its system-call interface, thereby allowing a user-space daemon program to provide an automated facility for creating and orchestrating sandboxed environments. This is the only feasible architecture on a general-purpose kernel such as Linux. Alternatively, the kernel may treat every software component, including its own subsystems, as an entity to be wrapped in a sandbox. In that case, the concept of a process itself would have to satisfy all three virtualisation axioms. Examples of such operating systems include the seL4 *microkernel* - the first operating system to be formally verified as free of programming errors [Kle09], and Google's Fuchsia - described by Pagano, Verderame, and Merlo [PVM21].

We first consider operating system virtualisation using a user-space daemon program. In this architecture, an application, referred to as a *container*, is defined as an encapsulation of "[...] a software component and all its dependencies in a format that is self-describing and portable, so that any compliant container runtime can run it without extra dependencies, regardless of the underlying machine and the contents of the container" [Ini16, p. 1]. A *container runtime* is the user-space daemon program responsible for bringing this portable but static representation of an application into execution. In runtime, a container consists of a collection of processes that share a restricted view of the system's resources. For example, every container "believes" it has a dedicated network stack with its own network interfaces, routing tables and packet-forwarding rules. All processes in the container can access and manipulate that network stack, but no other process outside the container has that capability. The container runtime configures this invariant and the operating system enforces it by namespacing system resources. The Open Containers Initiative (OCI) [Ini15] has developed a runtime specification that standardises the operations a container runtime needs to support. Most importantly, it must allow an external process called a *container engine* to hook into the lifecycle of a container. The container engine can use these hooks to manage all the containers on a single host system. In addition, the engine can attach

network and storage to containers, thereby allowing processes in different sandboxes to share state and communicate with each other, if required. At the highest level of abstraction sits an orchestration platform that manages containers on multiple hosts by interacting with the container engine on each system. This platform constantly monitors node and container health and dynamically multiplexes workloads based on various system properties of the cluster as to ensure maximum application availability.

It is important to note that, by definition, the kernel is assumed to be trustworthy. It has full control of all hardware resources and can access and modify the execution environment of every process on the system. In other words, noninterference between the kernel and user processes is not guaranteed. Therefore, if the kernel is compromised, all processes on the system become untrustworthy. It follows that if a process compromises the kernel, it transitively interferes with all other processes on the system. Hardening the operating system by implementing various security features such as mandatory access control has been the primary approach for protection against such scenarios. However, the size of a monolithic general-purpose kernel is too large to adequately create a threat model that captures all possible vectors of attack. This problem is of particular concern to infrastructure providers whose entire business model revolves around consolidating hundreds of potentially malicious client applications on the same server, all of which share the same kernel and are allowed to directly interact with it via its system call interface.

Unlike hardware virtualisation, this architecture does not use hardware emulation as an isolation primitive. This means that shadow pages need not be maintained per virtual environment. Input-output operations need only traverse the kernel's stack without any address translations and with the additional performance benefit of direct memory access. As a result, the isolation overhead is lower compared to a virtual machine, which allows more applications to be consolidated onto a single server. Furthermore, guests do not boot up complete operating system images, which reduces start times and memory consumption. Priedhorsky and Randles [PR17] use containers in high-performance computing clusters to run user-defined compute jobs and show that the imposed performance penalties are, at most, negligible compared to vanilla processes that have no additional isolation. Felter et al. [Fel15] show the exact same thing and further conclude that the Docker container engine is resource-friendlier and faster than the Kernel Virtual Machine (KVM) when stressing the "memory, IPC, network and filesystem subsystems" Felter et al. [Fel15, p. 1] of the Linux kernel by running a database inside a virtual environment and evaulating its performance via the SysBench OLTP benchmark [ZK04].

Google's gVisor [Goo18] attempts to sustain the performance advantages of containers whilst introducing an additional isolation boundary between the kernel and each container. The authors implement a substantial portion of the system call interface in a user-space process called Sentry. Their dedicated container runtime calls out to Sentry instead of the kernel when issuing system calls.

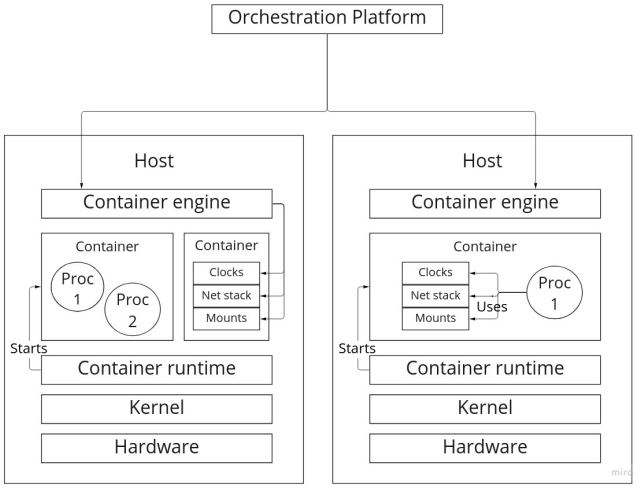

Figure 2.2: Operating system virtualisation architecture using containers. The container runtime starts containers on a single host. A user process can see a bundle of resources allocated to it by the kernel. The kernel guarantees that a process cannot see any other resources. The container engine manages all containers on a single system and allocates storage and networking to create explicit paths between containers. The orchestration platform talks to all engines inside a cluster to provide automatic workload management.

However, Young et al. [You19] show that network and memory allocation performance greatly suffer. This can be partially attributed to the fact that the Sentry process is implemented in a garbage-collected language and lacks the fine-grained optimisations contained in the Linux kernel. Agache et al. [Aga20] take a different approach and try to fuse the security of virtual machines with the performance of containers by programming a custom virtual machine monitor called Firecracker that runs on top of KVM. Firecracker completely relies on the Linux kernel for memory management, CPU scheduling and block I/O. To reduce its attack surface, the virtual machine monitor sacrifices portability by supporting a limited set of emulated network and block devices. To further strengthen the noninterference boundary, the devices have configurable built-in rate limiters that can control the number of operations per second, e.g disk/packets per second. Unlike a traditional container runtime, Firecracker's rate-limiting implementation does not rely on the kernel, which makes its isolation boundary to the kernel stronger. Anjali, Caraza-Harter, and Swift [ACS20] evaluate both gVisor and Firecracker and show that the latter "[...] is effective at reducing the frequency of kernel code invocations, but had a much smaller impact on reducing the footprint of kernel code" [ACS20, p. 12].

2.2 Processes

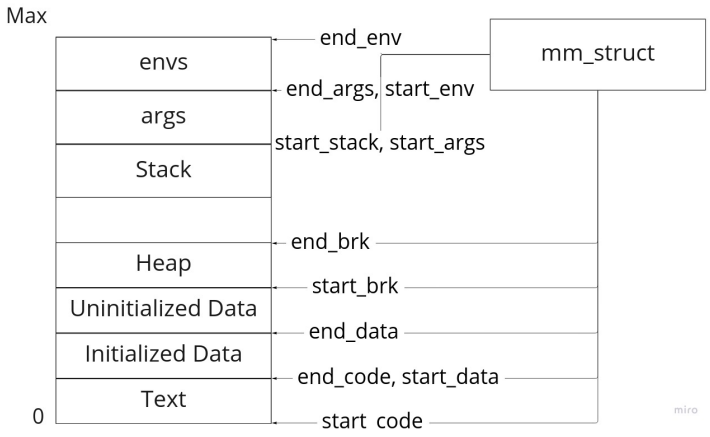

Figure 2.3: Memory layout of a process [SGG18]

A process is the fundamental abstraction of operating-system virtualisation. It is the primary unit of work in a computer system and represents a program in execution. This chapter introduces various aspects of a process - how it is laid out in memory (Chapter 2.2.1), how it is scheduled for execution (Chapter 2.2.2), and the system call interface for creating, terminating, monitoring and protecting processes from each other (Chapters 2.2.3, 2.2.4, 2.2.5, 2.2.7, 2.2.8).

Every process on Linux is represented as a `task_struct` [Tor22c] which contains various bits of information such as the process's memory descriptor, the table of open file descriptors, the running state of the process (see Table 2.1), its pending signals and so on. Processes are organised in a hierarchy, where each process can have an arbitrary number of children and a single parent. The kernel expresses this relationship by having each task store a pointer to its parent and a circular doubly linked list to all of its children.

2.2.1 Memory Layout

The kernel partitions the virtual address space of a process into multipe regions shown in Figure 2.3. The address space itself is encapsulated in a memory descriptor type called `mm_struct` [Tor22a] and every process is associated with one or more instances of that type. The text segment of the address space is a read-only memory section that contains the set of executable instructions that the program compiles down to [SGG18]. When the program is first scheduled for execution, the central processing unit jumps to the text section's initial memory address, which is stored in the `start_code` field of the process's memory descriptor and starts executing from there until it reaches the final linear address `end_code`. The data section contains globally defined variables that can be referenced throughout the entire lifecycle of the process [SGG18]. The data section is further deconstructed into uninitialised and initialised subsections. Similarly

to the text section, it does not grow or shrink because its size is known at program startup and does not change. The `start_data` and `end_data` fields in the memory descriptor refer to the contiguous memory region that represents the initialised data. The heap segment contains memory that is allocated during runtime as per demand. The process itself is responsible for releasing any memory acquired from the heap. The latter is important for two reasons. First, the heap grows upwards towards the stack and may overlap with it. Second, memory allocation interfaces such as `malloc` acquire memory from a system-wide pool. If a long-running process acquires dynamic memory but fails to release it, the whole system will eventually run out of memory - an example of spatial interference. The kernel tracks the heap areas allocated to a particular process via the `start_brk` and `brk` fields of the memory descriptor which denote its initial and final linear addresses, respectively. The stack is a data structure that acts as temporary storage for a process. Whenever a process calls a function, that function's parameters, local variables and return address are pushed onto the stack. When the function returns control to the caller, these values are popped of [SGG18]. The stack grows downwards towards lower addresses and its starting address `start_stack` sits right above the `main()` function's return address.

Note that the virtual address space of the process is sparse because it contains a hole between the stack and the heap that will require physical pages only if those two segments grow. This hole may be utilised to bring in page frames that can be shared between processes in order to reduce memory consumption and to allow two processes to communicate with each other. For example, the `libc` library is reentrant and can be brought into physical memory only once. The memory descriptor of a process that depends on `libc` does not have to load its own copy of the library into memory. Instead, its page table entries can map to the same physical frames that all other processes refer to when using the library. This reduces memory consumption and therefore has positive implications on same-host density.

2.2.2 Scheduling

Systems running more than one process rely on a process scheduler to determine, in real-time, which process should use the processing unit such that all processes do meaningful work and the core does not idle. The process scheduler maintains a queue of all processes that are currently being executed by a processing unit or are waiting to be assigned to one. This queue is represented in memory as a circular doubly linked-list of `task_struct` objects and is called the run queue because all of its tasks are in the `TASK_RUNNING` state. The kernel keeps a run queue local pointer `current` to the current task being executed by a processing unit. It is the scheduler's job to determine the most optimal sequence of processes to schedule from the run queue onto the processing unit in order to satisfy two conflicting requirements - low latency and high throughput. Low latency is crutial for processes that mostly perform input-output operations. Fetching data from a device, such as a network card, typically takes multiple CPU cycles. For this reason, the

scheduler moves a task currently waiting for an external event from the running queue into the wait queue and updates the `current` pointer to point to a different process. This operation is known as preemption. The wait queue consists of all processes currently waiting for data or a

State identifier	Description
`TASK_RUNNING`	The process is being executed by a CPU or is waiting to be assigned to one
`TASK_UNINTERRUPTIBLE`	The process is waiting for an event or a resource and cannot be interrupted by a signal while doing so
`TASK_INTERRUPTIBLE`	The process is waiting for an event or a resource and can be interrupted by a signal while doing so
`TASK_NEW`	The process has been created but it cannot be scheduled for execution and it cannot react to events
`EXIT_DEAD`	The process is being cleaned up and deleted
`EXIT_ZOMBIE`	The process has exited, the parent has been notified via a `SIGCHLD` signal but the parent has not reaped it yet

Table 2.1: Table of most important process states in Linux derived from [Tor22c] and [Tor22b]

signal. Their state is always either `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`. In order for an input-output bound process to make meaningful progress, it needs to be frequently rescheduled back onto the processing unit. Conversely, high throughput is important for processes that are primarily executing computations. These processes make the most progress by monopolising the processing unit. Hence, low-latency and high throughput are incongruous. Operating systems manage this incongruity by ranking processes based on their priority and allocating timeslices that determine how long a task can run until it is preempted.

On Linux, every `task_struct` is associated with a `sched_entity` object that holds task-related statistics used by the scheduler to do its job. The most important properties of this object are the `weight` and `vruntime` fields. The former represents a priority value that measures the task's willingness to be preempted. In other words, it represents how "nice" a process is to all other processes on the system. The value ranges from -20 to 19 with a default of 0. Larger values correspond to lower priorities and smaller values correspond to higher priorities. The `vruntime` (virtual runtime) field denotes the amount of time already allocated to the task. The invariant that the scheduler tries to enforce is that the process with the smallest virtual runtime should be the one that is currently using the processing unit. By definition, the process with the smallest virtual runtime is also the process that has used the processing unit the least. For this reason, the scheduler is said to be completely fair. Whenever the run queue is updated, either by an interrupt triggered by the system timer or a change in a task's state, the virtual runtime of the

task is incremented by a weighted delta

$$d = e \times w / \sum_{i=0}^{n} w_i \tag{2.1}$$

where e refers to the amount of time the task used the central processing unit since the last time the run queue was updated. w corresponds to the priority value of the task. n denotes the length of the run queue. The sum in the expression aggregates the priority values of all tasks in the queue. Hence, dividing w by the sum gives us a proxy of the task's priority relative to all other tasks in the queue. Recall that the scheduler monotonically increases the virtual runtime of the process by d. A process that performs a lot of input-output operations exhibits small processing unit bursts, hence e is smaller. Therefore, the virtual runtime for that process will increase more slowly and the process will be scheduled more frequently for execution. Similarly, if the process has a high priority, i.e w is small, the division in Equation 2.1 will result in a small fraction that will cancel out a big portion of e when multiplied.

The `sched_entity` object of every process is a part of a a red-black tree whose search key is set to the virtual runtime property. The scheduler picks a new process for execution by traversing the tree and finding the task with the lowest virtual runtime, which is always stored at the leftmost leaf node. The cost of traversal equals the depth of the tree, i.e it takes $\Theta(\log n)$. However, the scheduler caches the leftmost node, thereby reducing the cost of traversal down to $\Theta(1)$. Whenever a process transitions from the `TASK_NEW` state into the `TASK_RUNNING` state, it is assigned the minimal allowed virtual runtime. Therefore, it is automatically placed at the leftmost node and gets to execute immediately. As the system progresses, the process will shift more and more to the right branches of the tree and the rest of the tasks will get to execute.

2.2.3 Creation

The `libc` wrapper functions `clone()` and `fork()` are the only mechanisms for creating a new process from user space. The former is a non-portable but very flexible method for creating a duplicate of the calling process. It gives the caller fine-grained control over the resources that the parent and the child get to share. Both functions internally use the `SYS_clone` system call to create the child. However, `clone()` gives the caller the ability to configure the child's execution context, whereas `fork()` does not. Instead, `fork()` transparently preconfigures the child. It is important to note that the the configuration options passed into `SYS_clone` define the noninterference boundary between a child and its parent. As a matter of fact, the noninterference boundary is what allows us to differentiate between a process and a thread. A child whose memory descriptor refers to the same virtual memory pages as its parent is a thread. A child with separate virtual memory pages is a process.

```c
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *stack, int flags, void *arg, ...
         /* pid_t *parent_tid, void *tls, pid_t *child_tid */ );

/* Returns thread id of child on success, -1 on error with errno set */

#include <unistd.h>

pid_t fork(void);

/* Returns process identifier of child in parent, or -1 on error; Returns 0 in
    child */
```

Code snippet 2.1: Clone and Fork System Call Wrappers

Whenever a new child process is created, both the parent and the child share the same text segment. The child obtains a dedicated copy of the parent's stack, heap and data segments if and onl if it attempts to write to one of those segments. Otherwise, the kernel will not explicitly copy anything, due to the expensivness of the operation. This technique is known as copy on write.

`clone()` creates a child process that begins its lifecycle by executing the function pointer `fn`. The return value of `fn` represents the exit code with which the child process will terminate. The function pointer expects a generic argument as input that can be concretised by the caller via the `arg` parameter. The `flags` argument represents a bit mask used by the caller to configure the child's execution context. For example, setting the `CLONE_VM` flag will result in both processes sharing the same virtual address space. Memory writes performed by one process will be visible to the other process. Similarly, the `CLONE_FILES` flag can be set so that the child and parent processes share the same file descriptor table. If one of the processes closes a file descriptor, the other process is also affected. The bit mask can also be used to put the child process in freshly-allocated resource namespaces - an important noninterference construct discussed in Chapter 2.2.8. The lower byte of the bit mask is used to specify the child's termination signal. If left unset, the parent has no way of reaping the child.

The caller is expected to provide a memory region to be used as the new process's stack. The system call wrapper pushes the function pointer onto the stack and transfers execution to the kernel. The kernel creates the new process by copying the old process - taking into consideration which parts should be copied based on the `flags` bit mask. Importantly, the kernel places the start address of the user-defined stack into the stack register of the processing unit. Afterwards, the process is placed in the run queue and control is returned to the system call wrapper.

The system call itself returns two times - once in the parent process where the return value corresponds to the process identifier in the child, and once in the child process where the return value is zero. The system call wrapper handles the latter case by popping the function pointer of the stack and calling it [Sou22], thereby using the function pointer as the main entry point into the child process.

There are multiple caveats that need to be considered when calling `clone()`. First, the `stack` parameter must point to the top address of the stack. Because the stack grows downwards, the address to be passed to `clone()` is not the start address of the memory region but the end address, which needs to be aligned according to the application binary interface. Second, careful attention must be paid when creating a child with the `CLONE_VM` flag set. The child process cannot depend on anything defined in `libc` because it risks polluting the parent's runtime environment. In particular, important variables allocated in thread-local storage by the linker are not set up for the child, e.g the `errno` variable, the standard input-output streams and more. Instead, they are shared between parent and child.

Example 1

Code snippet 2.2 shows how to create a child process via the `clone()` wrapper. First, the parent prints its process identifier to standard output. Afterwards, it sets up a memory region to be used as the child's stack by allocating two pages of memory that can be read from and written to by both processes. The memory region is not backed by external storage and changes made by one process are not visible to the other. Furthermore, the starting memory address is page-aligned and chosen by the kernel. On line 22, the system call wrapper is executed with the `child_process` function set as the entry point into the child. The `shared_resources` variable refers to the bit mask that controls what the parent and child get to share. It is set to zero, indicating that they will not share any resources. The bit mask is ORed with `SIGCHLD` - the signal that will be sent to the parent when the child finishes execution. On lines 27-31, the parent blocks until it receives the `SIGCHLD` signal from the child and exits. In (2.3) the output of the program shows the relationship between the child and the parent obtained by calls to `getpid()` and `getppid()`.

If we were to modify the example in (2.2) so that the parent and child share the same virtual memory pages, then thread-local variables would not be properly placed in thread-local storage, potentially resulting in undefined behavior when interacting with `libc`.

The `fork()` system call creates a child process transparently preconfigured to share no resources with the parent and to respond to the `SIGCHLD` signal upon termination. In other words, it is equivalent to calling `clone()` with the `flags` bit mask set to zero and ORing it with `SIGCHLD`. `fork()` returns two times - once in the parent and once in the child. In the latter case, the returned value is zero. Hence, the caller can define the child's logic via a conditional statement.

```c
/* The entry point into the child */
int child_process(void *data)
{
    printf("Child: %d\nChild parent: %d\n", getpid(), getppid());
    return 0;
}

int main(int argc, char *argv[])
{
    printf("Parent: %d\n", getpid());

    size_t stack_len = sysconf(_SC_PAGESIZE)*2;
    int protection_flags = PROT_READ | PROT_WRITE;
    int configuration_flags = MAP_ANONYMOUS | MAP_PRIVATE | MAP_STACK;
    void *stack = mmap(NULL, stack_len, protection_flags,
                       configuration_flags, 0, 0);
    if (stack == MAP_FAILED)
        goto err;

    /* The parent will share nothing with the child, because no flags are set */
    int shared_resources = 0;
    pid_t child = clone(child_process, stack + stack_len,
                        shared_resources | SIGCHLD, NULL);
    if (child == -1)
        goto err;

    int status;
    if (waitpid(child, &status, 0) != child)
        goto err;

    return 0;
err:
    printf("%s", strerror(errno));
    return 1;
}
```

Code snippet 2.2: os-concepts/src/clone.c

```
$ ./clone
Parent: 329638
Child: 329657
Child parent: 329638
```

Code snippet 2.3: os-concepts/src/clone.c output

2.2.4 Termination

```c
#include <stdlib.h>

noreturn void exit(int status);

int atexit(void (*function)(void));

/* Returns 0 if successful; otherwise a nonzero value */

#include <unistd.h>

noreturn void _exit(int status);
```

Code snippet 2.4: Exit System Call and Wrappers

Processes can terminate abnormally or normally. Abnormal termination happens when a process receives a signal from the kernel or another process. For example, division by zero causes the hardware to detect a fault condition and notify the kernel. The kernel can then send a termination signal to the process that made the arithmetic error. Sending a signal to a process directly interrupts its execution flow by saving the current state of the registers and causing the processing unit to jump to a signal handler - a function dedicated to processing a particular signal. After the signal handler is executed, the process either terminates or continues its execution from the point of interruption.

A process terminates normally when it invokes the `_exit()` system call. In doing so, it notifies the kernel that it can reclaim all of the process's resources, including its memory descriptor, file table, scheduling information and more. Additionally, the kernel notifies the parent about the child's termination by sending it the termination signal defined when the child was created - typically the `SIGCHLD` signal. The child can send its termination status to the parent by passing a signed integer to the `_exit()` functon. Conventionally, an exit status of 0 means that the child has successfully executed its operations.

Alternatively, a process can call the `exit()` wrapper defined in `libc`, which internally uses `_exit()`. In addition to releasing operating-system resources, the `exit()` function also invokes a set of exit handlers defined by the application programmer and `libc` to cleanup auxillary resources. For example, `libc` defines an exit handler that flushes the standard input-output buffers of the process. Users can register exit handlers via the `atexit` function. Exit handlers are called in reverse order of registration. If an exit handler fails to return, then the remaining set is not called.

Example 2

```
1  void exit_handler_1(void)
2  {
3      printf("Process %d invoked exit handler 1\n", getpid());
4  }
5  void exit_handler_2(void)
6  {
7      printf("Process %d invoked exit handler 2\n", getpid());
8  }
9  int main(int argc, char *argv[])
10 {
11     printf("Parent process: %d\n", getpid());
12     /* Register exit handlers in parent */
13     atexit(exit_handler_1);
14     atexit(exit_handler_2);
15     pid_t child = fork();
16     if (child < 0)
17         goto err;
18     if (child == 0) {
19         /* Child process begins execution here */
20         printf("Child process: %d\n", getpid());
21         exit(0);
22     }
23     int status;
24     if (waitpid(child, &status, 0) != child)
25         goto err;
26     printf("Child exited with status %d\n", status);
27     return 0;
28 err:
29     printf("%s", strerror(errno));
30     return 1;
31 }
```

Code snippet 2.5: os-concepts/src/exit-normal.c

```
$ ./exit
Parent process: 14862
Child process: 14863
Process 14863 invoked exit handler 2
Process 14863 invoked exit handler 1
Child exited with status 0
Process 14862 invoked exit handler 2
Process 14862 invoked exit handler 1
```

Code snippet 2.6: os-concepts/src/exit-normal.c output

Code snippet 2.5 shows an example of normal process termination. The parent registers two exit handlers that print the identifier of the process that invoked them. On line 19, the parent creates a child via the `fork()` system call. The child prints its process identifier and invokes `exit()` with a status of zero. This causes the kernel to send a notification signal to the parent who is waiting at line 30 until it receives the signal. The parent prints the status of the child and exits. In (2.6) the program is executed. Notice that the exit handlers are invoked four times - two times in the parent and two times in the child. That is because a child inherits a copy of its parent's exit registrations. Since the child exits first, it always invokes the exit handlers before the parent. If the child had terminated via `_exit()`, then the exit handlers would have been executed solely by the parent.

Example 3

Code snippet 2.7 shows an example of abnormal process termination caused by an interactive interrupt signal. The parent creates a child that registers a signal handler to be executed whenever it receives a `SIGINT` signal. The signal handler is registered via the `sigaction` system call, which overrides the default handler for `SIGINT` with the one specified in the `.sa_handler` field. The signal handler prints a message to standard output and terminates the process. After registering the handler, the child suspends itself, i.e it manually enters the `TASK_INTERRUPTIBLE` state by calling `pause()`. On line 27, the parent waits for the child to exit. If the child does not receive a `SIGINT` signal, the parent will wait indefinitely. The program is executed in (2.8) by starting the parent as a background task and manually sending the `SIGINT` to the child process through the `kill` command. The signal handler that terminates the child is triggered and the kernel notifies the parent of the event. The parent reports the child's termination status and exits.

It is important to note that calling `printf` from a signal handler may be unsafe in certain circumstances. Consider a thread of execution being interrupted by a signal handler while it is executing `printf`. If the signal handler also calls `printf`, it will corrupt the internal data structures used to keep track of the amount of data and the current position in the buffer. This will lead to unexpected results when control is returned back from the signal handler to the thread. For this reason, the `printf` function is said to be async-signal unsafe, i.e it is not atomic with respect to signal interrupts. In the current example, the child does not call `printf` after the new signal handler is registered, so its usage is safe. In practice, however, processes are large and complex enough for this to become a problem.

Also, signal handlers cannot be used to reliably track the number of signals generated, because signals are not queued. While a signal handler is executing and the same signal is sent multiple times to the process, that signal is marked as pending and is later redelivered only once.

```c
/* Will get triggered when the caller pushes Ctrl+C */
void default_interaction_handler(int signal)
{
    printf("Terminating process\n");
    _exit(0);
}
int main(int argc, char *argv[])
{
    pid_t child = fork();
    if (child < 0)
        goto err;
    if (child == 0) {
        printf("Child identifier: %d\n", getpid());
        /* Child registers sigint handler */
        struct sigaction action = {.sa_handler = default_interaction_handler };
        if (sigaction(SIGINT, &action, NULL) != 0)
            goto err;

        /* Child is suspended until it receives a signal */
        pause();
    }
    /* Caller will block indefinitely until child exits */
    int status;
    if (waitpid(child, &status, 0) != child)
        goto err;
    printf("Child exited with status %d\n", status);
    return 0;
err:
    printf("%s", strerror(errno));
    return 1;
}
```

Code snippet 2.7: os-concepts/src/exit-abnormal.c

```
$ ./exit-abnormal &
[1] 162043
Child identifier: 162044
$ kill -s SIGINT 162044
Terminating process
Child exited with status 0
[1]+ Done ./exit-abnormal
```

Code snippet 2.8: os-concepts/src/exit-abnormal.c output

2.2.5 Monitoring

```
#include <sys/wait.h>

pid_t wait(int *status);

/* Returns process identifier of terminated child; -1 on error */

pid_t waitpid(pid_t pid, int *status, int options);

/* Returns process identifier of child; 0; -1 on error */
```

Code snippet 2.9: Wait System Call and Wrappers

The parent process often wants to gather information about when and how a child has terminated. This functionality is provided by `wait()` and `waitpid()`, as already shown in (2.2), (2.6) and (2.8).

2.2.6 Communication

2.2.7 Resource Management

2.2.8 Security

Chapter 3

Appendix A

List of Figures

List of Tables

Source Code Content

References

[ACS20] Anjali, Tyler Caraza-Harter, and Michael M. Swift. "Blending Containers and Virtual Machines: A Study of Firecracker and GVisor". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 101–113. ISBN: 9781450375542. DOI: 10.1145/3381052.3381315. URL: https://doi.org/10.1145/3381052.3381315.

[Aga20] Alexandru Agache et al. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[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.

[Esx06] Vmware Esx. "Performance Evaluation of Intel EPT Hardware Assist". In: 2006.

[Fel15] Wes Felter et al. "An updated performance comparison of virtual machines and Linux containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[Kle09] Gerwin Klein et al. "SeL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[Kue17] Simon Kuenzer et al. "Unikernels Everywhere: The Case for Elastic CDNs". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 15–29. ISBN: 9781450349482. DOI: 10.1145/3050748.3050757. URL: https://doi.org/10.1145/3050748.3050757.

[Lv12] Hui Lv et al. "Virtualization Challenges: A View from Server Consolidation Perspective". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. VEE '12. London, England, UK: Association for Computing Machinery, 2012, pp. 15–26. ISBN: 9781450311762. DOI: 10.1145/2151024.2151030. URL: https://doi.org/10.1145/2151024.2151030.

[Man17] Filipe Manco et al. "My VM is Lighter (and Safer) than Your Container". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 218–233. ISBN: 9781450350853. DOI: 10.1145/3132747.3132763. URL: https://doi.org/10.1145/3132747.3132763.

[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.

[PR17] Reid Priedhorsky and Tim Randles. "Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126925. URL: https://doi.org/10.1145/3126908.3126925.

[PVM21] Francesco Pagano, Luca Verderame, and Alessio Merlo. "Understanding Fuchsia Security". In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 12.3 (2021), pp. 47–64.

[Ran20] Allison Randal. "The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers". In: *ACM Comput. Surv.* 53.1 (Feb. 2020). ISSN: 0360-0300. DOI: 10.1145/3365199. URL: https://doi.org/10.1145/3365199.

[SGG18] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 10th. Wiley Publishing, 2018. ISBN: 9781119320913.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105.

[WR12] Carl Waldspurger and Mendel Rosenblum. "I/O Virtualization". In: *Commun. ACM* 55.1 (Jan. 2012), pp. 66–73. ISSN: 0001-0782. DOI: 10.1145/2063176.2063194. URL: https://doi.org/10.1145/2063176.2063194.

[You19] Ethan G. Young et al. "The True Cost of Containing: A gVisor Case Study". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotcloud19/presentation/young.

Online References

[Goo18] Google. *gVisor*. 2018. URL: https://github.com/google/gvisor (visited on 05/26/2022).

[Ini15] Open Container Initiative. *Open Containers Initiative Website*. 2015. URL: https://opencontainers.org/ (visited on 05/26/2022).

[Ini16] Open Container Initiative. *The 5 principles of Standard Containers*. 2016. URL: https://github.com/opencontainers/runtime-spec/blob/main/principles.md (visited on 05/26/2022).

[Sou22] Sourceware. *Glibc clone.S*. 2022. URL: https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/unix/sysv/linux/aarch64/clone.S;hb=HEAD (visited on 06/23/2022).

[Tor22a] Linus Torvalds. *Linux Memory Mapping Types*. 2022. URL: https://github.com/torvalds/linux/blob/master/include/linux/mm_types.h (visited on 06/23/2022).

[Tor22b] Linus Torvalds. *Linux Process Scheduling Core*. 2022. URL: https://github.com/torvalds/linux/blob/master/kernel/sched/core.c (visited on 06/23/2022).

[Tor22c] Linus Torvalds. *Linux Process Scheduling Header*. 2022. URL: https://github.com/torvalds/linux/blob/master/include/linux/sched.h (visited on 06/23/2022).

[ZK04] Peter Zaitsev and Alexey Kopytov. *Sysbench OLTP*. 2004. URL: https://github.com/akopytov/sysbench (visited on 05/26/2022).