Friedrich-Alexander-Universität
Technische Fakultät

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Paul Bergmann

Design and Implementation of User-, PID-, Network- and Mount-Namespaces in JITTY-OS

Bachelorarbeit im Fach Informatik

30. August 2022

Please cite as:
Paul Bergmann, “Design and Implementation of User-, PID-, Network- and Mount-Namespaces in JITTY-OS”, Bachelor’s Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, August 2022.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

Design and Implementation of User-, PID-, Network- and Mount-Namespaces in JITTY-OS

Bachelorarbeit im Fach Informatik

vorgelegt von

Paul Bergmann

geb. am 30. Juni 1999
in Nürnberg

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dr.-Ing. Volkmar Sieh**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**1. Mai 2022**
Abgabe der Arbeit:	**30. August 2022**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Paul Bergmann)
Erlangen,
30. August 2022

ABSTRACT

Virtualization has always been an important task of operating systems. Over the last two decades, cloud computing has enjoyed ever-increasing popularity. This has led to the widespread use of virtualization approaches such as hypervisors and virtual machines. One approach that has become particularly popular is virtualization at the operating system level, also known as containerization. Namespaces are the basic building block for OS-level virtualization on Linux.

JITTY-OS is an operating system with the goal of being simple and understandable while maintaining compatibility with Linux. To support operating system level virtualization in JITTY-OS, this bachelor thesis is concerned with designing an implementation of namespaces for JITTY-OS. Specifically, user, PID, network and mount namespaces are implemented. Thereby, no forwarding of mount events takes place in mount namespaces. It should also be possible to easily support other types of namespaces in the future.

To accomplish this, it first examines behavior which is shared by all kinds of namespaces as provided by Linux. Then it describes the structures used to implement the different types of namespaces in JITTY-OS and justifies the choices made. The result of this thesis is a functional implementation of namespaces, which is equivalent to Linux in its semantics.

KURZFASSUNG

Virtualisierung war schon immer eine wichtige Aufgabe von Betriebssystemen. Über die letzten zwei Jahrzehnten hinweg erfreute sich Cloud Computing immer wachsender Beliebtheit. Das hat zur weitreichenden Verwendung von Virtualisierungsansätzen wie Hypervisors und virtuellen Maschinen geführt. Ein Ansatz der sich besonders großer Beliebtheit erfreut, ist Virtualisierung auf Betriebsystemebene, auch bekannt als Containerisierung. Namensräume sind auf Linux der Grundbaustein für Virtualisierung auf Betriebsystemebene.

JITTY-OS ist ein Betriebssystem mit dem Ziel, simpel und verständlich zu sein und dabei gleichzeitig Kompatibilität zu Linux zu erhalten. Um Virtualisierung auf Betriebsystemebene in JITTY-OS zu unterstützen, beschäftigt sich diese Bachelorarbeit mit dem Entwurf einer Implementierung von Namensräumen für JITTY-OS. Konkret implementiert werden User-, PID-, Network- und Mount Namensräume Dabei erfolgt in Mount Namensräumen keine Weiterleitung von Einhängeereignissen. Außerdem soll es möglich sein, in Zukunft weitere Arten von Namensräumen einfach zu unterstützen.

Dazu untersucht sie erst Verhalten die auf Linux von allen Namensräumen geteilt werden. Anschließend beschreibt sie die Strukturen die zur Implementierung der verschiedenen Arten von Namensräumen in JITTY-OS verwendet wurden und begründet die getroffenen Entscheidungen. Das Ergebnis dieser Ausarbeitung ist eine funktionsfähige Implementierung von Namensräumen, welche in in ihrer Semantik äquivalent zu Linux ist.

CONTENTS

1 INTRODUCTION

Operating systems (OSs) have long had the task of distributing a computer's resources to applications [Sal66]. Ideally, it appears to each application as if the computer is available to it alone. On the other hand, applications also want to interact with each other. This is the core problem of virtualization. There must be a balance between isolation and sharing.

This development has intensified over the last two decades. With the rise of cloud computing, various virtualization technologies are gaining popularity. This includes both hypervisors and techniques such as containerization [Ber14]. Containerization in particular provides a flexible approach to virtualization for many applications, balancing between isolation and sharing [Sol+07].

Namespaces provide the basic building blocks for containerization on Linux [Ros16]. Every kind of namespace isolates a certain resource between different applications. Because applications decide which namespaces they share and which they don't, they are more or less isolated from each other. This has seen use even outside of cloud computing. Desktop applications use these lightweight virtualization facilities to provide increased security [DER22]. The systemd init system[1] in particular allows to isolate services using various kinds of namespaces.

Support for lightweight virtualization is thus an important feature for modern OSs. While Linux provides many such features, it is also very complex. In January 1, 2020, the Linux kernel contained nearly 28 million lines of code and configuration, spread out over 66 thousand files [Lar20]. The JITTY-OS project, developed at the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), aims to implement a Linux-compatible OS in a clean and understandable manner. The project additionally aims to use its own just in time (JIT) compiler FAUCCC [Sie+20].

JITTY-OS is currently able to boot and run a Debian userspace up to Debian 6. Newer Debian versions ship with systemd as the mainly used init system. The new init system expects the kernel to support namespaces as provided by Linux. This was previously not the case in JITTY-OS. The goal of this thesis is thus to implement user-, PID-, network and mount namespaces to allow the usage of new systemd and Debian versions. It also provides a framework for the future implementation of different kind of namespaces.

Namespaces are a far-reaching kernel feature. They virtualize a lot of different resources and are thus used in a lot of different kernel systems. With the main goal of JITTY-OS being simplicity, it is especially important for the namespace implementation to not increase complexity too much.

To achieve this goal, this thesis will first introduce relevant OS concepts and the semantics of various resources found on Linux and JITTY-OS. It will then present the structures used to implement namespaces for these resources and give rationale for the decisions taken. Afterwards, the implementation's simplicity is evaluated.

[1] https://www.freedesktop.org/wiki/Software/systemd/

Note that while Linux supports additional kind of namespaces, support for them is not sensible in JITTY-OS. These include inter-process communication (IPC) and control group (cgroup) namespaces in particular. Most of the IPC primitives virtualized by these namespaces are not supported in JITTY-OS. Neither are cgroups. The implemented namespaces should be sufficient to run new versions of Debian and systemd.

2 FUNDAMENTALS

Before going in depth into the design and implementation of various kinds of namespaces in JITTY-OS, this chapter will give an introduction to the relevant operating system concepts. Firstly, Section 2.1 explains both general operating system concepts and concepts of JITTY-OS specifically. Afterwards, Section 2.2 introduces virtualization as one of the major features of an operating system. The subsections following Section 2.3 introduce the various interfaces provided by many operating systems and how they are virtualized using namespaces.

2.1 Operating Systems

The primary purpose of a computer system is to run application programs. What this application entails depends on the specific problem being solved. Different computer systems provide wide variety in their underlying hardware architecture. This variety spans their instruction set architecture (ISA), input/output (IO) devices, the bus structures used to access these devices and much more. How these structures have to be programmed differs widely across different hardware setups. It is not sensible to expect application programmers to handle all the intricacies and idiosyncrasies of these setups. According to Tanenbaum, the first major purpose of an operating system is to provide abstractions over these different hardware setups to application programs [Tan15, Section 1.1.1].

2.1.1 Operating System Components

To provide these abstractions, operating systems feature many different components. An operating system's central component is the *kernel*. How exactly the kernel controls hardware depends on the specific piece of hardware. However, generally speaking, many pieces of hardware provide consumable resources in some form. A network interface provides the packets it has received. A break beam sensor provides information once it detects motion. When the kernel waits for such resources to become available, it can do so using two general methods. It can wait *actively*, doing nothing, but staying active on the central processing unit (CPU) while waiting. However, this is a waste of computing resources.

The other approach is waiting *passively*. This involves sleeping until resources are available. The piece of hardware then *interrupts* the procedure's sleep. An interrupt handler, installed by the kernel, gets executed and fetches the requested resources. Afterwards, the normal procedure can continue. Since no work is performed until the resources are available, computing power can be saved or used for other tasks.

Interrupts usually signal important events and should be handled quickly. Most architectures do not allow nested interrupts. Servicing new interrupts is disabled during their handling. However,

depending on the event, a single interrupt handler can take a long time. For example, when handling network interface interrupts, the handler must fetch data from the device, parse it and forward it to the correct subsystem where it will be processed further. Many operating systems thus partition interrupt handling into two parts: a prologue and an epilogue. The prologue is the classical interrupt handler. Prologues should be short and only perform necessary interaction with the hardware. Further processing happens during the epilogue. Epilogues can themselves be interrupted by prologues, leading to a short interrupt latency.

Many computer systems run untrusted software not directly provided by the system. If left unattended, such software can hijack the system. Most architectures thus feature two or more *privilege levels*. Only procedures running on an elevated privilege level in *supervisor* mode may control the hardware directly. Most applications instead run unprivileged in *user* mode. In most operating systems, the only procedures running in supervisor mode are those in the kernel. Of course, most applications still want to access the underlying hardware. To support this, the kernel provides a set of *system calls*. When an application issues a system call, a change in privilege level occurs and a procedure previously installed by the kernel starts executing. Since the elevated procedure is provisioned by the kernel, not by an untrusted application, it can access the hardware in a restricted manner.

Operating systems provide other interfaces to their internal structures as well. A prominent example for this is the `proc(5)` file system provided by Linux. By reading the files provided by the file system, applications can retrieve statistics both about the general system and about other applications specifically. For example, a task manager can use the file system to list applications running on the system and display the program they are running.

2.1.2 Introducing JITTY-OS

Since operating systems have to run on such a wide variety of hardware, they often contain a lot of optimizations for these different varieties as part of their source code. An example of this is the `memcpy` function in Linux. There are multiple implementations of this function for different ISAs. Furthermore, even on the same ISA, different versions of a processor may support different features allowing further optimizations. Thus, this task of *instruction selection* (in this case: selecting the most efficient `memcpy` implementation) falls to the operating system developer [Pet15].

Having these kinds of optimizations as part of the source code leads to more cognitive overhead and violates a separation of concerns. In general, this kind of instruction selection is better handled by compilers and other automated systems. However, most of the the time, the operating system is compiled ahead of time and then distributed to its users. The user does not receive the source code but a binary directly fit for execution. In this case, the compiler can not know all the idiosyncrasies of the target platform the binary will later be executed on. It thus has to be conservative while optimizing; not every feature is supported on every platform. JITTY-OS aims to circumvent this problem by instead compiling the operating system just in time, that is, on the target platform, just prior to its execution. This gives the compiler knowledge about the exact target architecture and thus allows more specific optimizations. In the meanwhile, the operating system developers do not have to concern themselves as much with such optimizations. In fact, one of the main goals of JITTY-OS is to implement a clean and simple operating system. To allow integration with existing application programs, JITTY-OS strives to maintain both an application binary interface (ABI) and an application programming interface (API) compatible with Linux [Sie+20].

2.1.3 Multithreading

Most systems are not just running a single application. For example, users on a desktop systems may run a text editor, a web browser and a music player, all at the same time. Even in a single program, there may be multiple strands of execution. A web server may handle multiple connections in parallel. Each request is logically independent of every other. This is further amplified by modern hardware architectures. Instead of increasing single core performance, modern computer systems achieve an increase in overall computing power by converting more and more to highly parallel multi-core architectures [HS12, Chapter 1].

In this thesis, such parallel strands of execution are referred to as *threads*. In many texts, threads only refer to parallel strands belonging to the same program. Different strands belonging to different programs are often referred to using other names, for example as processes or thread groups. In the context of this thesis, this distinction is mostly unimportant and the term thread is used as an umbrella term.

Threads are able to start new threads. Every thread has a parent, which is usually the thread performing the start operation. To allow interaction between threads, the operating system provides various IPC mechanisms. Each IPC mechanism allows a thread to pass information on to its peers. These mechanisms include for example the ability to send signals, which signify simple events.

2.1.3.1 Synchronization

Parallelly running threads often communicate through shared memory. However, if one is not careful, these threads may interfere with each other and results are unpredictable. JITTY-OS thus supplies mechanisms fit for synchronization. The approach taken is simple, in keeping with JITTY-OS's effort towards simplicity. Through the use of locks, resource access in JITTY-OS happens mutually exclusive with regard to other threads. To acquire a lock, a thread must call `enter` on the lock instance. On the other hand, calling `leave` releases the lock. There are two kinds of locks employed in JITTY-OS. The first are spinning locks (s-locks). When a thread tries to acquire an s-lock while it is currently contested, it waits actively for the lock to be released. The other kind are waiting locks (w-locks). When a thread tries to acquire an w-lock while it is currently contested, the thread is marked as blocked and another runnable thread is dispatched instead. When the lock is released, one of the threads waiting for it is marked as runnable.

2.1.3.2 Thread Safety Analysis

To ensure the proper locks are acquired when accessing certain resources, JITTY-OS makes use of Clang thread safety analysis [HBS14]. This analysis alone is not ideal in an operating systems developement context. For example, Clang does not analyze, whether synchronization requirements are met when using function pointers. However, JITTY-OS makes heavy use of function pointers. Furthermore, Clang does not allow specifying synchronization requirements as logical disjunctions, only as conjunctions. JITTY-OS thus additionally aims to use its own compiler FAUCCC to provide thread safety analysis [Hei22].

To perform the analysis, annotations are added to variables and functions when declared. This thesis uses the same annotations to give an overview over synchronization requirements. If a variable is protected by a lock `mylock`, it is annotated with `GUARDED_BY(mylock)`. Variables that are not mutated after their initialization and thus require no further synchronization are annotated with `CONST_AFTER_INIT`. Sometimes, a lock has to be held across multiple function calls to uphold a synchronization constraint. When this happens, the lock must be acquired in an outer scope, before each function is entered. The relevant functions can then be annotated as `REQUIRES(mylock)`.

Whenever the function is called, `mylock` must already by held by the caller, allowing the function to access state protected by the lock.

To prevent deadlocks, JITTY-OS enforces a global order in which locks must be acquired. Locks belong to different levels. When a thread acquires a lock of a low level, it may no longer acquire locks of higher levels. Thus, if a thread wants to acquire both, the higher level lock has to be acquired first, before acquiring the lower level lock. Also, in general, threads may only acquire a single lock per level.

Similarly to how some functions should only be called when holding a certain lock, some functions should only be called from an epilogue or a thread context. A thread context is given if the procedure is related to a currently running thread. This happens for example when threads perform system calls. Some procedures, like retrieving the program of the currently running thread, only make sense when there *is* a currently running thread. Thus, JITTY-OS annotates functions with `THREAD_CONTEXT`, `EPILOGUE` and `PROLOGUE` to signify they should only be called in thread context, when servicing an epilogue or a prologue respectively.

2.2 Virtualization

The resources provided by computer systems are always limited. On the other hand, the quantity of requested resources is relatively unbounded. For example, the number of CPU cores on a typical desktop system may be 8 or 16. The number of running threads can easily reach 1000 in contrast. The situation is similar for other resources.

Isolation between threads is another concern, as systems may execute untrusted applications. For safety and stability reasons, resources used by one thread should not be accessible to other, untrusted threads. When threads intend to share resources, they can selectively lift the isolation intentionally. These two tasks, resource distribution and isolation, are the second major purpose of an operating system [Tan15, Section 1.1.2]. They can be understood as providing a virtual view on the resources given by the computer system to threads and are henceforth consolidated under the term *virtualization*.

Classically, only few resources are isolated between threads. This most notably includes the thread's virtual memory space. Many other resources are shared. Threads know of their mutual existence and share their knowledge of the underlying system: which hardware devices are available and their state of operation. Some fields of application require more separation. Take for example a server hosting service: customers expect their rented system to behave as if they had exclusive access to the hardware.

A common approach to fulfill this requirement is the use of *virtual machines*. A piece of controlling software, called the hypervisor, creates the illusion of multiple virtual sets of hardware on a single physical machine. Each of the virtual machines executes its own kernel. Since it is the kernel that provides a hardware interface to applications, and the kernel itself is only able to see the virtual hardware, applications running in different virtual machines experience a great deal of isolation between each other.

While this can be an advantage, it can also be a disadvantage. In some cases, applications seek to communicate with each other. Since little context is shared between the different virtual machines, more expensive communication primitives like the network have to be used to accomplish this. Isolating only the necessary resources allows using more efficient mechanisms when desired.

OS-level virtualization is an approach that tries to provide a middle ground between running a single kernel and using virtual machines. All applications use the same OS-kernel. However, they are more isolated than usual. For example, threads may not know of their mutual existence

and can not interact with each other in any way. One mechanism providing OS-level virtualization popularized by Linux are *namespaces*. Each kind of namespace takes a resource classically shared between threads and virtualizes it. This enables the selective isolation strove for above. OS-level virtualization has gained great popularity over the past years. It forms the basis for technologies like Docker and containerization in general [Ros16].

2.3 Namespaces

Namespaces virtualize distinct, previously global resources. For every namespaced resource, every thread is member of at least one namespace instance. Instead of seeing the global state, each thread only sees an isolated instance provided by the namespace. Namespace based virtualization is an opt-in feature, used mostly by frameworks like Docker, or applications that wish to isolate themselves, like web browsers. Each namespace has a default instance: the *initial namespace*. When the system boots, the first started thread is placed in the initial namespace. By default, threads inherit their parent's namespace membership. The namespace affiliation can change, either determined by the parent when creating, or by the thread itself during its runtime.

JITTY-OS strives to be compatible with Linux. The exact semantics of namespaces are thus mandated by Linux. The following subsections will each introduce a specific resource, how it behaves and how it is virtualized by namespaces.

2.3.1 Credentials and User Namespaces

On a Linux system, each thread has a set of associated credentials. These credentials encompass the thread's user ID (UID) and group ID (GID) [2]. They are simple integers. Credentials are used to authorize resource access. For example, every file has an associated owning user and owning group. When a thread tries to access a file, the file's credentials are compared to the credentials of the thread. If they match up, the thread is authorized to use the file. Threads are able to inspect their own UID and GID using the `getuid(2)` and `getgid(2)` system calls.

Threads with a UID of 0 are granted a special status: they are considered *privileged*. Privileged threads are allowed to access any resource, bypassing the aforementioned authorization step. Moreover, some operations are only available to privileged threads in the first place. Usually, this is the case for operations that affect some global state visible to other threads. For example, some signals sent between threads causes the receiving thread to terminate. Normally, the sender must have the same UID as the receiver to be authorized to do so. A thread with UID 0 can send such signals to any thread.

Credentials are global state, as there only is a single number space. Threads either have a UID of 0 in this space and are privileged, or they are not. User namespaces virtualize this number space. Each user namespace provides its own credential number space Threads are privileged *in a namespace*, if they have a UID of 0 *in that number space*.

However, being privileged in a user namespace is not useful on its own. User namespaces are only fully useful in combination with other kinds of namespaces. Every other kind of namespace has an *owning user namespace*. Whenever a thread tries to perform some privileged operation on the resources governed by a non user namespace, it must instead be privileged in the owning user namespace.

[2] Note, that on Linux and JITTY-OS, threads have many different associated UIDs and GIDs. For instance, these systems differentiate between *real* and *effective* UIDs. This is mostly irrelevant for the user namespace implementation, as all kinds of credentials are handled the same.

Not every resource is namespaced, leading to not every resource having an owning user namespace. Because of this, the classical notion of a UID and GID still exist. To avoid confusion, this thesis refers to these classic credentials as kernel user ID (KUID) and kernel group ID (KGID) in *kernel space*. Credentials can be translated between kernel space and user namespaces. The initial user namespace is special in this regard: it uses an identity translation to kernel space. Thus, KUID and KGID in kernel space and UID and GID in the initial user namespace have the same value.

Every time credentials are visible to a thread, they are translated into the user namespace the thread is a member of. Take for example the `getuid(2)` system call: the system call returns the thread's KUID mapped into its current user namespace. If a thread changes its user namespace, it returns a potentially different value afterwards. Vice versa, whenever a thread provides credentials, they are translated into the context of the receiver. For example, the `setuid(2)` system call allows changing a thread's KUID. When the thread calls the function, it provides a UID in its own namespace which is then translated into kernel space.

User namespaces form a hierarchy. At the time of its creation, every user namespace records the currently active user namespace as its parent. This is relevant when checking for privileges. As an additional condition, a thread is privileged inside a user namespace if it is privileged in any ancestor user namespace. Threads privileged in the initial user namespace are considered privileged in the classical sense. This condition ensures these threads are privileged in all other namespaces as well, as the initial user namespace is ancestor to every namespace.

2.3.2 Threads and PID Namespaces

In JITTY-OS, strands of execution are organized into two structures: threads and thread groups. Every such structure has a globally unique identifier. Threads are identified by their thread ID (TID). Thread groups are identified by their thread group ID (TGID). TID and TGID are the values returned by the `gettid(2)` and `getpid(2)` system calls respectively. Note, that there is some confusion between the terms TGID and process ID (PID). In Linux, the term PID is *internally* used to identify a single thread. To avoid confusion, this thesis uses JITTY-OS's definitions for TID and TGID. However, the namespaces are called PID namespaces in accordance with Linux.

Every thread is part of a thread group. Thread groups are said to terminate when their last member thread terminates. Thread groups build a tree hierarchy. A parent thread group may have multiple children thread groups. Every thread group has a parent except for the tree's root. Whenever a thread dies, the parent to its thread group bears the responsibility of making sure the child's resources are released.

The first thread in the root of this hierarchy is called the *init* thread. It is the first thread created when the system boots up. The init thread always has a TID of 1. Its thread group always has a TGID of 1. The init thread and its thread group are critical for system operation. If a parent thread groups terminates prior to its children, the children are re-parented to the init thread group. If the init thread group terminates the whole system halts.

Namespaces virtualize this thread hierarchy and the TID and TGID number spaces. Instead of identifiers being globally unique, they are only unique within a single PID namespace and may be reallocated in a different PID namespace. Every PID namespace has its own init thread. Whenever a thread group is orphaned it is re-parented to the init thread group of its PID namespace. The first thread to join a PID namespace becomes the namespace's init thread. Similarly to how the global init thread was seen as critical for the system's operation, the namespace init thread is seen as critical for the namespace's operation. When a namespace's init thread terminates, all other threads inside the namespace are terminated by a `SIGKILL` signal. Further attempts to join the namespace fail with the error `ENOMEM`.

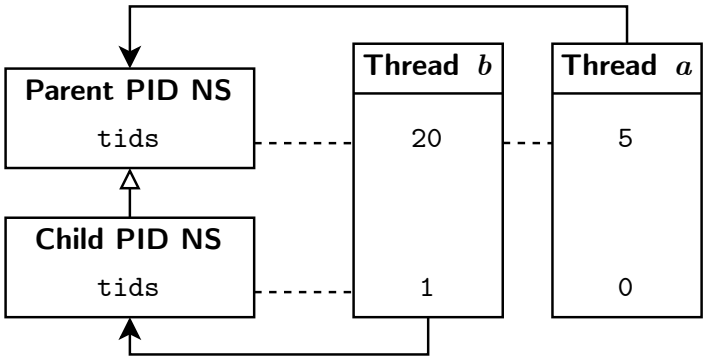

Figure 2.1 – An example showcasing the PID namespace hierarchy. Thread a is a member of the parent PID namespace and visible there with the TID 5. Since it is invisible in the child namespace, it has a TID of 0 in that namespace. Thread b is a member of the child namespace and thus visible in both. It has a TID of 20 and 1 respectively.

PID namespaces are arranged in a tree hierarchy as well. At the time of its creation, every PID namespace records the currently active PID namespace as its parent. Threads and thread groups can be *visible* in multiple namespaces in that hierarchy. This is illustrated in Figure 2.1. It is a common interface to return TIDs and TGIDs of 0 when a thread is invisible. For example, the global init thread has no parent. When that thread issues the `getppid(2)` system call to retrieve its parent's TGID 0 is returned instead. This interface is extended to PID namespaces. Threads can not see other thread's in parent PID namespaces. When such threads interact, identifiers in the parent are displayed as 0.

2.3.3 Networking and Network Namespaces

Networking plays an important role in modern computer systems and the availability of a network stack is essential to many applications. There are many different protocols and technologies used in networking. Their functionality ranges from discovering systems directly connected to the local machine up to providing secure connections to systems on the other side of the world. To both simplify their implementation and allow reuse, when different protocols are used for specific tasks, the network stack is usually arranged into layers. Each layer abstracts a certain aspect of network communication. This is the case in JITTY-OS as well [Sau20].

The network stack's lowest layer is the *data link layer*. It has to responsibility of managing the systems network interfaces. Network interfaces include Ethernet and Wi-Fi cards for example. In general, network interfaces allow communication over a local network with machines directly connected to the current system. The data link layer contains drivers for local network interfaces and provides functions to allocate and send packets over these devices. On the data link layer, systems are usually addressed using media access control (MAC) addresses.

Above the data link layer lies the *network layer*. An application's communication partner may reside on the other side of the world. The data link layer only allows communication with systems directly connected to the local machine across the local network. The network layer is introduced to overcome this limitation. A wider network, like the internet, is composed of many interconnected smaller networks, each allowing direct communication. The network layer has the task of finding a route from the local to the target machine across these smaller networks. The result of this process is

a host on the local network. That host is in turn connected to both the local and a different network and will forward the sent packet. On the network layer, systems are usually addressed using internet protocol (IP) addresses.

The different addressing schemes used in the network and data link layers lead to a problem: the network layer's routing process returns an IP address identifying the target host on the local network. However, the underlying link layer and its network interfaces can only address hosts using MAC addresses. The IP addresses thus have to be resolved to MAC addresses. The exact process on how this happens depends on the internet protocol version. JITTY-OS currently implements only version 4 and uses the address resolution protocol (ARP).

The data link and network layers have three caveats. Firstly, there is no guarantee that the sent packets will actually reach their destination. Packets can potentially be dropped by any intermediate link. Secondly, there is no guarantee packets arrive in the order they were sent in. They may overtake each other. Thirdly, both layers send data at packet granularity. Applications usually expect to read and write streams of bytes instead. Depending on the use case, at least some of these problems are unforgivable. To alleviate them, the *transport layer* is implemented on top. It uses protocols like the transmission control protocol (TCP) to provide stronger guarantees.

A common networking API provided to threads are *Berkeley sockets*. Its central component, the *socket*, represents the endpoint of a communication channel to a foreign system. Following the Unix philosophy, threads can communicate using these sockets by writing and reading streams of bytes through the `write(2)` and `read(2)` system calls. Additionally, information about different layers of the network stack is made available over the `proc(5)` file system. For example, `/proc/self/net/arp` contains information about recent ARP lookups.

Network namespaces virtualize and isolate the various components of these layers. Normally, all network interfaces are visible to all threads. With the introduction of network namespaces, each interface belongs to a certain namespace instead. Only threads within that namespace are able to see and use the interface. All interfaces initially belong to the initial network namespace and may be moved to a different namespace. When the namespace is released, the interfaces are moved back to the initial network namespace. The data structures used by the network layer for routing are usually shared across the whole system. Instead, they become per namespace state. The files in the `proc(5)` file system reflect these changes: they display information local to the network namespace of each thread instead of global state.

2.3.4 File Systems and Mount Namespaces

Computer systems store data on a wide variety of storage media, like hard disk drives (HDDs) and solid state drives (SSDs). Addressing data on such drives happens on a relatively low level of abstraction, for example using block addresses. Applications expect a higher level of abstraction, usually addressing data as files and directories. In between lies a layer mapping between files and directories and the block addressing of the underlying storage media: the file system. There is a wide range of possible concrete file systems like `ext4`, `btrfs` and the network file system (NFS), each with its own advantages and disadvantages.

These different concrete file systems organize themselves in vastly different and generally incompatible ways. Applications should not have to worry about the concrete file system underlying the data they access. To accomplish this, most Unix-like operating systems, including Linux and JITTY-OS, introduce another layer of indirection between file systems and applications: the VFS. In the VFS, files are organized in a single unified directory hierarchy. Different concrete file systems are made available in the hierarchy by *mounting* them using the `mount(2)` system call.

On a system without namespaces, the unified directory hierarchy is shared between all threads. Every mount operation by every thread is visible to every other thread. With the introduction of mount namespaces, the directory hierarchy is instead shared only between threads belonging to the same mount namespace. When a mount operation is performed in one namespace, it does not automatically become visible in other namespaces. Moreover, this allows mounting different concrete file systems at the same mount point as long as these mount operations occur in different mount namespaces.

Linux additionally allows specifying a *propagation type* for mounts. This allows to share certain selected mount operations even across mount namespaces. This is not implemented in JITTY-OS. Even without support for propagation, the implementation still supports a wide variety of applications. For example, Docker defaults to disabling propagation during its mount operations [Sta17].

2.4 Related Work

Other Unix-like systems provide facilities enabling OS-level virtualization as well. This notably includes jails as provided by the FreeBSD project. Jails behave roughly similar to a combination of user- and PID namespaces. The process environment, including visible processes, the file system and network resources are isolated between different jails. Each jail is bound to a single IP address. Additionally, file access is restricted in the style of the `chroot(2)` system call [KW00]. Other examples include Solaris zones [TC04].

Suites commonly used for OS-level virtualization on Linux, like the open container runtime [Bou16] and Docker, require additional operating system support apart from namespaces. Namespaces virtualize but they do not provide an upper limit on the amount of system resources threads may use. For this purpose, OS-level virtualization software on Linux uses cgroups [Ros16].

Threads still share context because not every resource is virtualized. If these threads are supposed to observe the system as if they were completely isolated, access to some resources should be prohibited. In particular, system calls that mutate global state should be disabled. On Linux, this is possible through the `seccomp(2)` system call.

3 ARCHITECTURE

The following chapter will first give an introduction to the concerns and shared structures when introducing any kind of namespace virtualization to the operating system in Section 3.1. The following sections then discuss the specific challenges in the implementation of User-, PID-, Network- and Mount Namespaces and present and discuss how these challenges are solved in JITTY-OS.

3.1 General Considerations

There are a multitude of aspects that need to be considered when designing a namespace implementation, regardless of the virtualized resource. There is quite some shared behavior any kind of namespace follows. This section will first introduce how namespaces in general behave from the point of view of userspace threads. This behavior poses certain constraints on how these namespaces must be implemented. The section will then present how JITTY-OS complies with these constraints.

3.1.1 Behavior Exposed to Userspace

The behavior userspace threads are exposed to includes both how threads are able to create new namespace instances and how these instances behave afterwards.

Linux provides two main methods to construct new namespace instances: using the `clone(2)` system call, during a thread's creation, and using the `unshare(2)` system call, during a thread's runtime. The implementation thus needs to provide functions to create new namespace instances and make them accessible for these system calls.

When a thread wants to access a namespaced resource, compared to earlier, additional steps are necessary. The resources are no longer available as a single static instance in a single static location. Instead, they are part of the namespace instance. The thread thus must be able to access the namespace instance it is a member of. Since accessing resources is a potentially frequent operation, this access must be computationally inexpensive.

Furthermore, a thread's namespace affiliation may be inspected and manipulated using the `proc(5)` file system. For each thread with TID `tid`, there exists the `/proc/[tid]/ns` subdirectory. This subdirectory contains a file for every type of namespace, for example `/proc/[tid]/ns/user`. These files serve three purposes:

- They uniquely identify the namespace the thread is a part of through the `st_dev` and `st_ino` fields as returned by the `stat(2)` system call. If these fields match up for the namespace files of two different threads, these threads are in the same namespace. The namespace implementation thus must be able to uniquely identify different namespace instances.

- They allow threads to discover further information about the namespace using the `ioctl(2)` system call. Take for example the tree structure spanned by PID namespaces: the `ioctl(2)` system call may be used to retrieve the parent of a PID namespace. The kernel must thus be able to retrieve this information when queried. Furthermore, it must be able to do so in the `proc(5)` file system implementation.

- They allow threads to change their namespace affiliation using the `setns(2)` system call. The kernel must thus be able to use the new namespace for subsequent resource accesses.

In general, threads are able to change their namespace affiliation during their runtime. PID namespaces pose an exception in this regard: The PID namespace of a running thread can not be changed. This restriction is in place since if it were possible, it would change what a thread thinks to be its own TID. This would break many applications and libraries relying on the assumption that the TID never changes. Instead, `unshare(2)` and `setns(2)` allow a thread to specify the PID namespace its children will be placed in. This can only be done once in the lifetime of a thread. The kernel thus must track this namespace intended for children, make sure it is only set once and pass it to child threads once they are created.

Not only threads can be affiliated with a certain namespace instance. Namespaces may be referenced by multiple different kernel systems. For example, the file descriptors retrieved by opening one of the files in `/proc/[tid]/ns` reference a certain namespace as well. All these references are *owning*; they extend the lifetime of the namespace being pointed at. The kernel must thus provide functions to create owning references to a namespace. The various systems that may reference a namespace must be able to use these functions. The kernel must also provide functions to release the resources associated with a namespace once there are no owning references left. Since most of the time, the same systems that create references also release them, the releasing functions must be accessible to these systems as well.

Every non user namespace has an *owning user namespace*. When a thread tries to perform a privileged operation on a resource governed by a non-user namespace, the system behaves differently compared to a non-namespaced system: if the thread is privileged in the owning user namespace, it is allowed to perform the operation, even if it is unprivileged in other parts of the system. The kernel must be able to handle this: whenever a resource is accessed, the corresponding non-user namespace and its owning user namespace must be available. Also, it must be possible to check whether a thread is privileged in a user namespace at many points since namespaces cover a wide range of systems.

3.1.2 Necessary Structures

The following section will describe how the wanted behavior described in Section 3.1.1 is achieved in JITTY-OS.

3.1.2.1 High Level Overview

Firstly, each type of namespace is contained in its own module. Where this module lies in the dependency chain depends on the type of namespace, since different types of namespaces depend on different kernel systems. It would be possible to put every function of every type of namespace into the same module. However, the different types of namespaces virtualize mostly unrelated resources and their functions thus are incohesive. The taken approach has the advantage of separating concerns.

Every namespace module provides functions to manage the life cycle of a namespace. There are `ns_*_unshare`, `ns_*_dup` and `ns_*_release` functions for every kind of namespace, creating a new instance, creating a strong reference to an existing instance and releasing a strong reference to an existing instance respectively. Another approach would be to inline these functions every time they are used. However, all of them are used in numerous places. Inlining them would lead to a lot of duplicated code, hindering the ability to maintain the functions. The approach using shared functions is thus advantageous. These functions are suitable for use in the `clone(2)` and `unshare(2)` system calls, as long as the namespace module is available in their implementation.

To correctly track the lifetime of namespace instances and implement owning references, reference counting is employed. With this scheme, every namespace instance holds an integer field called the reference counter. When the namespace instance is first created using one of the `ns_*_unshare` functions, the reference counter is set to one. Whenever a strong reference to the namespace is created using one of the `ns_*_dup` functions, the reference counter is incremented by one. When a reference is released through one of the `ns_*_release` function, the reference counter is decremented by one. When the reference counter reaches zero, its managed resources are no longer needed and are subsequently released.

Reference counting is not without caveats. It is an easy mistake to forget incrementing or decrementing the reference counter at the right place, leading to hard-to-fix memory leaks and use-after-free bugs. Another possible solution would be to employ garbage collection. With this scheme, resources which are no longer in use are found and released by an automatic garbage collection system. While JITTY-OS provides a garbage collector, it is unsuited for the namespace use case. The JITTY-OS garbage collector does not support associating resources with a function that should be run when that resource is released. However, this is necessary for some kinds of namespaces that need to perform additional cleanup in their release routine. For example, network namespaces need to move network interfaces to the initial network namespace during their destruction.

To be able to provide an identifier for the `proc(5)` file system, every namespace instance holds an `id` field. The field is populated when the namespace is created using a simple ID allocation system. The system keeps track of allocated IDs in a simple bitset and uses a linear scan to search for the first free new ID. More sophisticated allocation setups are possible. For example, Linux uses a hash map to store used IDs. However a simple linear scan approach is sufficient for JITTY-OS and easy to understand and implement. It is in principle also possible to use the namespace instance's address as identifier. This approach has two drawbacks. Portability is limited, since the address may not fit into the `st_ino` field as returned by the `stat(2)` system call. Secondly, it is a security concern since it leaks an address used by the kernel into userspace.

3.1.2.2 Concrete Structures

Figure 3.1 illustrates the concrete structures used in JITTY-OS. The structs `thread` and `thread_group` were used by JITTY-OS to manage threads and thread groups respectively prior to this thesis. All other structures are new additions. Since the reference counter `refcnt` and identifier `id` are required for every kind of namespace, they are stored in a shared base class `ns_base`. This avoids copying the fields to the derived classes and thus avoids code duplication. The identifier is only written to once when the namespace is created and read-only afterwards. It is thus marked as `CONST_AFTER_INIT`. The same namespace instance may be used at several points in parallel. Because of this, the reference counter may be written to and read parallelly. Accesses thus have to be synchronized. This is achieved using a simple mutex `refcnt_lock`. It would also be possible to use atomic variables for the reference counter, bypassing the need for locking. However, using atomic variables for reference counters is uncommon in JITTY-OS.

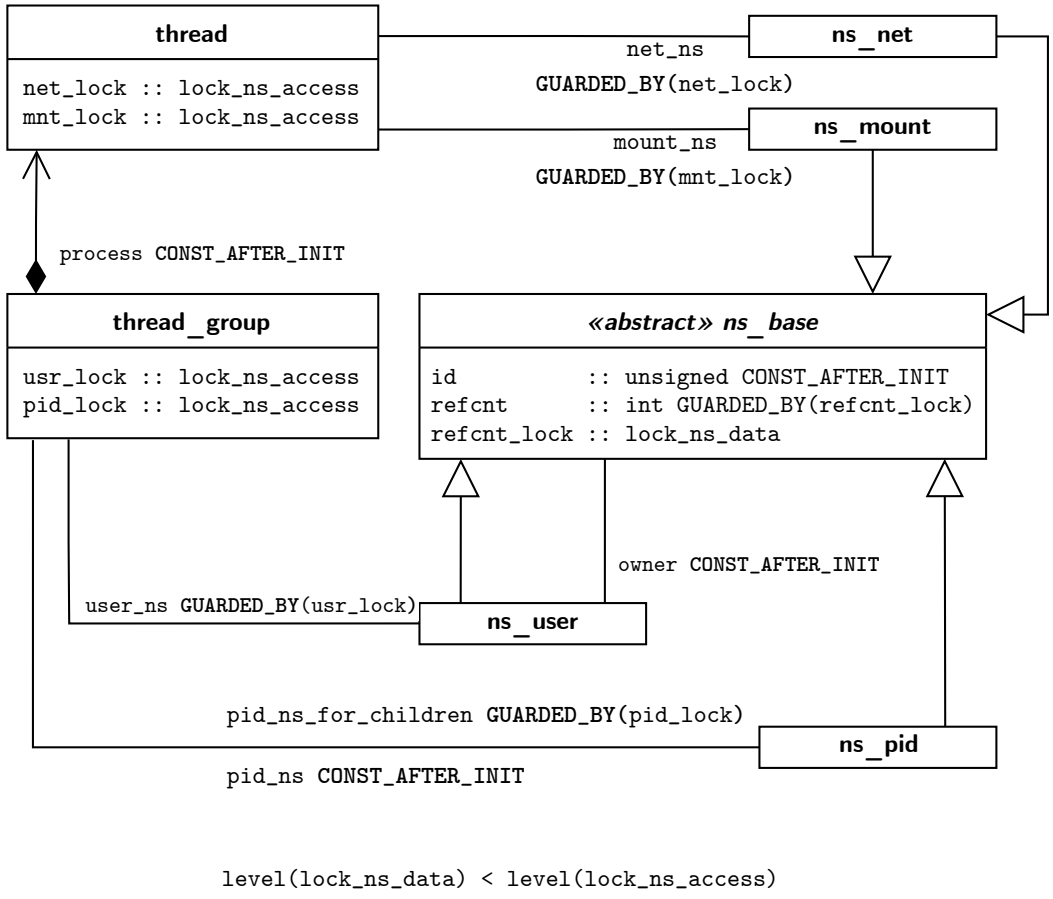

Figure 3.1 – UML diagram describing how structures shared between different namespace types are implemented in JITTY-OS and how each of them is connected to the thread control block. The structures required for each namespace type individually are described in depth in their corresponding sections.

To be able to reference a non user namespace's owning user namespace, the shared base class contains `owner`, an owning reference to a user namespace. This achieves the behavior desired in Section 3.1.1. Whenever a thread accesses a namespaced resource it has to hold a reference to that namespace; otherwise it would not be able to access that resource in the first place. The owning user namespace needed for privilege checking is now simply retrievable through the stored reference. Since the owning user namespace does not change during the lifetime of a non user namespace, the reference is marked `CONST_AFTER_INIT`.

Threads must be able to access the namespaces they are a member of and associated resources. To achieve this, each thread stores a set of strong references in its thread control block. It would also be possible to store this reference the other way around. With this different scheme, there would be a global list of namespace instances, each containing a list of its member threads. This would make it easy to find all threads inside a namespace. This operation is fairly uncommon. On the other hand, this different scheme would make the common operation of finding associated resources for a thread expensive. Keeping a reference from each thread to its namespaces is thus preferable.

Threads in multithreaded thread groups are not allowed to change their user- and PID-namespace affiliation. The threads in a thread group thus always belong to the same user- and PID-namespaces. To indicate this, the references to a thread's user- and PID-namespaced are stored on the thread group instance instead. This incurs additional cost when retrieving the namespace instance, since there is another level of indirection. However, the cost should be negligible. On the other hand, storing the reference on the thread group more accurately models the semantic of the namespaces being shared across the thread group.

A thread's namespace references may be accessed by multiple threads in parallel. Take for example a thread a with TID `tid`. This thread can change its network namespace affiliation using the `setns(2)` system call. Thread a will access `a->net_ns`. It will first release the old reference by decrementing the reference counter. Afterwards, the reference is replaced with a new one. At the same time, another thread b may retrieve a's network namespaces by opening `/proc/tid/ns/net`.

(a)

Thread a	Thread b
	`tmpb = a->net_ns`
`tmpa = a->net_ns`	
`tmpa->refcnt--`	
`free(tmpa)`	
	`tmpb->refcnt++`

(b)

Thread a	Thread b
	`enter(a->net_lock)`
	`tmpb = a->netns`
`enter(a->net_lock)`	`enter(tmpb->refcnt_lock)`
▪	`tmpb->refcnt++`
▪	`leave(tmpb->refcnt_lock)`
▪	`leave(a->net_lock)`
`tmpa = a->net_ns`	
`enter(tmpa->refcnt_lock)`	
`tmpa->refcnt--`	

Figure 3.2 – A race condition when accessing `a->net_ns`, stemming from the fact that dereferencing twice and incrementing a counter does not happen atomically on most platforms. On JITTY-OS, it is solved using mutual exclusion.

(**a**) On most platforms, the operation `a->net_ns->refcnt++` actually happens in two steps. The result of `a->net_ns` has to be copied into a temporary variable. The race condition occurs when the thread is preempted during these two steps.

(**b**) The race condition illustrated in **(a)** is solved using mutual exclusion: thread a has to wait for b to finish to be able to access the reference. Thus, b already owns an reference to the namespace before a is able to release the namespace.

During the opening process, *b* will call `ns_net_dup` on the reference `a->net_ns`, incrementing the reference counter. Thus, there are parallel write and read accesses to the same variables `a->net_ns` and `a->net_ns->refcnt`. In particular, accessing the reference and modifying the reference counter do not happen in a single atomic step on most architectures. There is need for synchronization; otherwise, the situation illustrated in Figure 3.2 (a) may occur: when *b* starts reading the old reference, it may be preempted by *a*. Thread *a* releases the old reference, decrementing the reference counter. At that point, the reference counter may reach zero, leading *a* to release the resources belonging to the old namespace instance. Afterwards, *b* resumes while still holding a reference to the now released namespace, leading to a use-after-free. Retrieving the reference and incrementing the reference counter has to happen in a single atomic step.

In JITTY-OS, this is achieved by mutual exclusion through a set of locks on the namespace references as illustrated in Figure 3.2 (b). There is one lock per type of namespace since they are unrelated to each other. It is not possible to protect the reference counter and the reference itself with the same lock; the namespace, including its reference counter, may be used independently of the thread. To be able to retrieve the reference and increment the reference counter in a single step, threads must be able to acquire the reference counter lock while already holding the reference lock. The locks protecting the namespace references thus have to reside on a higher level than the lock protecting the reference count. Using atomic instructions instead of locks is actually not an option in this case: most contemporary architectures do not support dereferencing a reference and incrementing a number in a single atomic instruction. Using transactional memory is a possible option. However, this too is uncommon in JITTY-OS.

As lined out in Section 3.1.1, accessing the namespace references to retrieve associated resources is a frequent occurrence. It is worthwhile to optimize this operation. The race condition presented in the previous two paragraphs only occurs when the namespace reference is written to and replaced with a different reference. A thread has exclusive permission to modify its own namespace references. They are never written to by someone else. This is true even for User- and PID-namespaces which are shared among multiple threads in a thread group. In this case, they may only be modified when there is just a single thread in the thread group, guaranteeing exclusive access. A thread thus may read its own namespace references without acquiring the corresponding reference lock. This prevents the thread from having to wait for the lock to be uncontested.

3.1.3 Complications when Virtualizing Resources

A resource may be available as a singular, static instance. When this resource is namespaced, multiple instances of the resource are made available. This is, after all, on of the purposes of virtualization. The resources are released as soon as the namespace is released. Other parts of the system may rely on the static lifetime of resources. If this is the case, caution has to be taken. For example, previously only a single network loopback device with static lifetime existed in JITTY-OS. The networking code using the device relied on the fact, that the device would never be released. With the introduction of network namespaces, each namespace creates its own loopback device instance. This instance is released when the namespace is released. The networking code, in turn, had to be updated. In this case it now uses an owning pointer to the loopback device.

3.2 User Namespaces

User namespaces fulfill a special role in the namespaced system: every non-user namespace has a corresponding owning user namespace. Which threads are privileged regarding resources governed

by that non-user namespace depends on the owning user namespace. They are thus a good fit to be implemented first. Section 3.1.2 described how user namespaces embed into the rest of the system. The following will now present the operations required from user namespaces themselves and the structures used to implement them in JITTY-OS.

3.2.1 Operations on User Namespaces

There are three main operations performed in a user namespace. Firstly, credentials may be translated. Whenever two systems in different user namespaces interact, the credentials have to be translated at some point. There are a variety of interfaces that transmit UIDs and GIDs between foreign user namespaces. They can be partitioned into three categories:

1. Interfaces translating from kernel credentials into a thread's user namespace. These include the `getuid(2)` and `stat(2)` system calls for example. The results of these system calls visible to the thread are always valid in the context of the thread's user namespace.

2. Interfaces translating from a thread's user namespace into the context of the kernel. The `setuid(2)` system call is an example for this type. Compared to the previous point, credentials are now translated in the opposite direction. The input parameters given by a thread are valid in the context of the thread's user namespace.

3. Interfaces that let threads in different user namespaces communicate. For example, threads are able to transmit credentials across Unix domain sockets using the `SCM_CREDENTIALS` ancillary message. Hereby the sender provides a UID and a GID valid in its own user namespace. When the receiver reads the message, the sent credentials are translated into its context.

Translations are required for a wide range of interfaces and should thus be implemented efficiently.

The second main required operation involving user namespaces, is the ability to check whether a thread is privileged inside a user namespace. This is actually an extension of the first operation: among other conditions, a thread is privileged inside a user namespace if its KUID maps to 0 inside the namespace. This illustrates the need for an efficient translation operation. There are other conditions as well. In particular, checking for privileges involves walking a user namespace's ancestry chain. Additionally, the creator of a user namespace is granted additional privilege. The implementation should be usable in a wide variety of contexts; every other kind of namespace depends on this.

In contrast, user namespaces themselves have nearly no dependencies. To be able to allocate their structures, they require the ability to allocate both memory and identifiers. Furthermore, the implementation should be simple. Apart from that they are self contained.

Lastly, threads need to be able to create translation tables for UIDs and GIDs once they create a new user namespace. This is made possible through the `proc(5)` file system. To create a new mapping for its user namespace, a thread with TID `tid` may write to `/proc/[tid]/uid_map` and `/proc/[tid]/gid_map` respectively. The mappings may only be initialized once. The translation structures should thus be optimized for reading, not for writing.

3.2.2 Structures Overview

The data structures used to implement user namespaces in JITTY-OS are shown in Figure 3.3. The main class representing user namespaces is `ns_user`. Since it shares behavior with other kinds of namespaces it inherits fields from `ns_base` as described in Section 3.1.2. This includes the `owner` field. As opposed to non user namespaces, user namespaces do not have an owning user namespace.

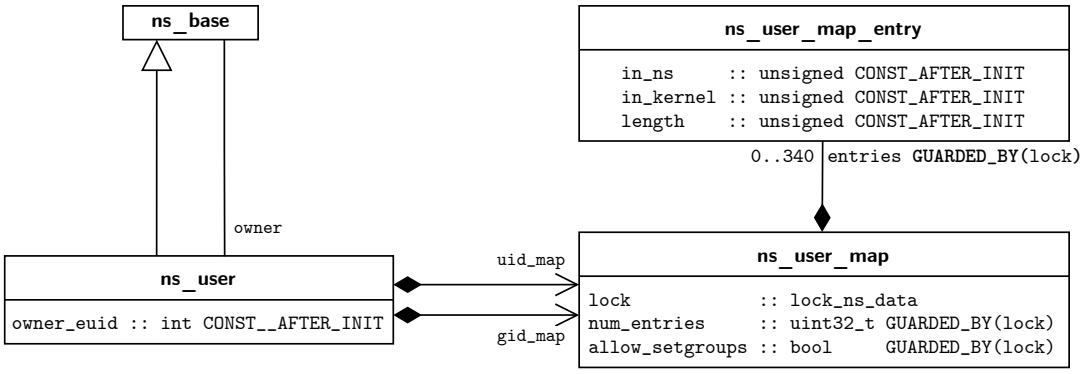

Figure 3.3 – The structures used to implement user namespaces. The shared structure `ns_base` is described more in depth in Section 3.1.2 and Figure 3.1. The limit of 340 entries per `ns_user_map` instance is given by Linux. With this limit, `sizeof(struct ns_user_map) <= 4096` holds, allowing the structure to fit onto a single page on most platforms.

However, they have a parent user namespace. To avoid having to store an extra field and leaving the `owner` field unused instead, the `owner` is simply used to store a reference to the parent.

The `ns_user` instance has to store the effective KUID of its creator in the `owner_euid` field as it is later required for privilege checking. The field is not modified after the namespace's creation and can be marked `CONST_AFTER_INIT`.

The `ns_user_map` and `ns_user_map_entry` classes are the heart of the user namespace implementation. Their purpose is to make translating credentials possible. User namespaces have distinct mappings for UIDs and GIDs but the translation process is exactly the same for both. To avoid duplicating the structure and associated functions, only one class is used. Instead, `ns_user` holds two different instances, one for each type of credential.

The map stores the `allow_setgroups` boolean. It controls whether threads inside a user namespace are allowed to perform the `setgroups(2)` system call. This restriction was put in place by Linux because of a security vulnerability in earlier versions of the namespace interface [Ker15]. This boolean is only needed once per user namespace, the other version remains unused. It remains on the GID mapping because of the semantic closeness to group credential operations. In principle, it may be stored on the user namespace instead. However, note that mutations to the `allow_setgroups` flags and writing the GID mapping have to happen atomically with respect to each other. Mutating the flag is not permitted anymore once the mapping is set up. Thus, a procedure mutating the flag first has to check whether a mapping already exists before setting the flag. The procedure could be interrupted between these two steps were it non-atomic, leading to a disallowed state. In JITTY-OS, this achieved using mutual exclusion. Using transactional memory is a possibility in this case, however this is uncommon in JITTY-OS.

3.2.3 Credential Translation Tables

JITTY-OS uses the `ns_user_map` and `ns_user_map_entry` structures shown in Figure 3.3 to translate credentials between different user namespaces. Each `ns_user_map` contains an array of `ns_user_map_entry`. There are a maximum of 340 entries per map. This limit is given by Linux. It ensures the size of a `ns_user_map` instance is not larger than 4096 bytes. It thus fits on a single page on most platforms.

3.2.3.1 Data Format

In the following, only UIDs are referred to but the argumentation is exactly the same for GIDs. Every entry in the translation table establishes a mapping between two intervals of credentials, respectively

$$[\texttt{in\_ns}, \texttt{in\_ns} + \texttt{length}]$$

and

$$[\texttt{in\_kernel}, \texttt{in\_kernel} + \texttt{length}]$$

With this entry, the consecutive ascending UIDs between `in_ns` up to `in_ns` + `length` inside the namespace are mapped to the consecutive ascending KUIDs between `in_kernel` up to `in_kernel` + `length` in kernel space. Take for example the entry given in Listing 3.1. The UID 0 inside the namespace is mapped to KUID 1000 in kernel space. Similarly, the UID 1 inside the namespace is mapped to KUID 1001 in kernel space.

Mapping an ID known to be in a certain interval is possible in constant time. The procedure has to calculate the offset between the ID and the start of the source interval and apply that offset to the start of the target interval. Finding the correct interval is the bulk of the work. It will be shown this is possible in linear or even logarithmic time in the number of intervals. Storing translation tables like this is thus sufficiently efficient and simple to understand.

Storing intervals mapping between kernel space and a user namespace allows translations between the two to be performed in a single step. For example, during the `stat(2)` system call, one translation step would be taken; the `stat` structure's UIDs and GIDs are stored in kernel space and then translated directly into the caller's user namespace during the system call. To translate credentials between different user namespaces, a detour has to be taken. Because there is no direct mapping, they first need to be translated from the source namespace into kernel space and then into the target namespace afterwards. Note, that when a thread creates a new user namespace and sets up the new mapping, from the thread's perspective the mapping is between the newly created namespace and the parent user namespace. Thus, to store the new mapping when it is written, it has to be translated from the parent user namespace into kernel space.

It would also be possible to instead store a mapping between the namespace and its parent user namespace. This would speed up the creation of new mappings as they could be stored directly, without further translation. However, every other credential translation would be slowed down as a result. In the aforementioned `stat(2)` example, every ancestor of the caller's namespace would have to be retrieved. The credentials would then be translated from kernel space into the oldest ancestor. Afterwards, they would be translated into the second oldest ancestor, and so on, until the caller's namespace is reached again. Since translating credentials is a frequent operation compared to setting up the mappings, this approach was not taken.

3.2.3.2 Translation Procedure

Actually translating the credentials between the user namespace and kernel space is a three step process, illustrated in Listing 3.2. The shown listing translates an ID `id` from kernel space into a

```
1 struct ns_user_map_entry sample = {
2   .in_ns = 0, .in_kernel = 1000, .length = 2
3 };
```

Listing 3.1 – A sample entry to a user namespace's translation tables to illustrate ID mapping

```
1  int translate_id_into(struct ns_user_map *map, int id) {
2    /* Step 1 - Check whether map is initialized */
3    enter(map->lock);
4    bool init = map->num_entries > 0;
5    leave(map->lock);
6    if (! init) return OVERFLOW_ID;
7
8    /* Step 2 - Find the matching entry */
9    assume_enter(map->lock);
10   struct ns_user_map_entry *entry = matching_entry(map, id);
11   assume_leave(map->lock);
12   if (! entry) return OVERFLOW_ID;
13
14   /* Step 3 - Translate ID */
15   int offset = entry->in_kernel;
16   return entry->in_ns + offset;
17 }
18
19 struct ns_user_map_entry *matching_entry_into(struct ns_user_map *map, int id)
20     REQUIRES(map->lock) {
21   for (int i = 0; i < map->num_entries; ++i) {
22     struct ns_user_map_entry *entry = &map->entries[i];
23     if (id >= entry->in_kernel && id < entry->in_kernel + entry->length) {
24       return entry;
25     }
26   }
27   return NULL;
28 }
```

Listing 3.2 – The algorithm used to translate credentials between a user namespace and kernel space with optimized locking. A simple linear search is used to find the interval the ID is covered by.

user namespace. However, the algorithm works the same in the other direction, just all occurrences of `in_kernel` are replaced with `in_ns` and vice versa.

The first step is given from line 2 up to line 6. Credentials can only be translated if the mapping is already initialized. If the map is uninitialized, the special `OVERFLOW_ID` value is returned, as mandated by Linux. [3] The mapping may only be read of the corresponding lock is held. However, in the implementation the lock is only really acquired during this first step and bypassed afterwards. This is allowed since the mapping may only be initialized once. If a thread observes the mapping to be already initialized, it can safely assume it will never change afterwards. However, it is still necessary to synchronize with other threads trying to initialize the mapping. To do so, the lock is acquired at the beginning just to check this condition.

This is a trade-off: on the one hand, it allows multiple readers in parallel. On the other hand, correctness is slightly more complicated to reason about. The lock would also be a candidate to be implemented as a readers-writer lock. However, JITTY-OS does not provide an implementation for this, by design. The most efficient lock implementation should instead be chosen by the surrounding infrastructure.

In the second step, the algorithm finds the entry spanning the interval in kernel space in which `id` is contained, that is, that fulfill $\texttt{id} \in [\texttt{entry->in\_kernel}, \texttt{entry->in\_kernel} + \texttt{entry->length}]$.

[3] The default value is 65534. On Linux, this is configurable. This was not needed for JITTY-OS and was left out for simplicity's sake.

That entry is later used to find the corresponding ID inside the namespace. In JITTY-OS, the entry is searched for by simply iterating through the map entries linearly as shown in lines 19 to 28. It is also possible to use binary search to find the matching entry. There is a caveat to this approach: it requires sorting the entries. When translating into the namespace, the ID is compared to the `in_kernel` values. The entries should thus be sorted by the `in_kernel` values. When translating into kernel space though, the entries should instead be ordered by their `in_ns` value, as the ID is compared to that field instead. Thus when using binary search, the mappings must either be stored twice, or the algorithm has to resort to linear search for one of the two translation directions. To strive for simplicity, JITTY-OS uses linear search in both cases. This approach is sufficiently efficient and easy to understand.

The third and last step, implemented in line 14 up to 16, actually translates the ID. Each entry covers a consecutive range of credentials in kernel space. The offset between the first covered ID and the ID to be translated if calculated. That offset is then used to find the corresponding ID in the consecutive interval inside the namespace.

3.2.3.3 Reading in new Mappings

```
acquire(map->lock)
if map->num_entries > 0 then
  release(map->lock)
  return Error: Already initialized
end if
num_read := 0
while there are lines left to read do
  Read and parse line
  Translate from parent namespace into kernel space and store into map->entries[num_read]
  for i := 0 to num_read − 1 do
    if map->entries[i] overlaps with the new entry then
      release(map->lock)
      return Error: Overlapping entry
    end if
  end for
  num_read := num_read + 1
end while
map->num_entries := num_read
release(map->lock)
```

Algorithm 3.1 – Simple quadratic algorithm to read in ID mappings into the `struct ns_user_map` instance `map`.

The main concern when reading in new credentials mappings is synchronization. Since the ID mapping is made public over the `proc(5)` file system, multiple threads may try to write it parallelly. Additionally, synchronization with mapping readers is required; they should not observe inconsistent states. To achieve this, JITTY-OS uses the algorithm given in Algorithm 3.1. The lock protecting the table is hold during the entire operation. Otherwise, two threads could observe the mapping to be uninitialized and try to initialize it at the same time, leading to an inconsistent state. In JITTY-OS, a s-lock is used. Since writing the mapping potentially takes a long time, a w-lock can be viable

instead; this would keep other threads from waiting actively, allowing the CPU to be used for other tasks. However, this complicates providing a global order of locks.

Every line provided by the thread contains a mapping between credentials in the newly created namespace and its parent namespace. To be able to store the mapping, the ID inside the parent namespace has to be translated into kernel space. Furthermore, the specification disallows the written intervals with each other. This is the same problem described in Section 3.2.3.2: the algorithm has to find an interval in which the new ID is contained in. Because of the same considerations as described above, a linear search is used. This leads to algorithm as a whole being quadratic in the number of lines. However, reading in mappings is an infrequent operation and the approach taken is simple in contrast.

3.2.4 Privilege Checking

With the ability to map credentials into a namespace being implemented, checking whether a thread is privileged inside a namespace is relatively straightforward. There are three possible conditions for a thread to be privileged in a namespace a:

1. A thread is privileged in a if it is a member of a and its KUID is mapped to the UID 0 inside a.
2. A thread is privileged in a if it is a member of a's parent and the thread's KUID matches the KUID of a's creator.
3. A thread is privileged in a if it is privileged in any of a's ancestors.

These conditions are implemented as shown in Listing 3.3. The first two conditions are directly translated in line 3 and line 8 respectively The third condition allows some variation. To find whether the thread is privileged in any ancestor, the algorithm follows the ancestry chain and checks the first two conditions for every chain element. This algorithm could also be implemented recursively, however this is avoided to keep the function call stack small. The approach taken has no external dependencies and uses just the namespace structures. This allows the function to be used in a wide variety of contexts, as requested in Section 3.2.1.

```
1  bool is_current_privileged_in(struct ns_user *user_ns) {
2    for (; user_ns != NULL; user_ns = user_ns->base.owner) {
3      if (ns_user_current() == user_ns
4       && translate_id_into(user_ns->uid_map, current()->euid) == 0) {
5        return true;
6      }
7
8      if (ns_user_current() == user_ns->base.owner
9       && user_ns->owner_euid == current()->euid) {
10       return true;
11     }
12   }
13
14   return false;
15 }
```

Listing 3.3 – The algorithm used to check whether the current thread is privileged inside the user namespace `user_ns`. The `current()` and `ns_user_current()` functions retrieve the currently running thread and its user namespace respectively.

3.2.5 Transmitting Credentials between different User Namespaces

There are a multitude of IPC primitives that transmit credentials between threads in potentially different user namespaces. These come in two variants: primitive, where the sender does not know which thread exactly will receive the message, and those where the sender does know.

Unix domain sockets are an example for the former primitive type. Threads may send their UID and GID through the `SCM_CREDENTIALS` ancillary message. However, Unix domain sockets are accessed through a path name. This path name may be visible to threads in differing user namespaces. The credentials must thus be translated into the receiver's user namespaces. The sender generally does not know which thread opened the socket and will receive the message. It is thus impossible to translate the credentials at the sender site. When a sender sends credentials through the socket, JITTY-OS first translates them into kernel space. Later, when the receiver comes to retrieve the sent data, they are translated from kernel space into the receiver's user namespace. It would also possible to transmit the credentials in the sender's user namespace alongside a reference to the sender's user namespace. This is however error prone as it requires correctly keeping track of another reference to the sender's namespace. Also, there is little gain since it still requires translating the credentials twice.

Signals are an example of the latter primitive type. The signal is always addressed to a thread or a specific group threads, known to the sender. When the receiver it handles the signal, it receives a `struct siginfo` instance. The sender's UID is stored in the `si_uid` field of that instance. Since the receiver is known, the sender could in principle translate the UID while filling the `struct siginfo`. However, this leads to a race condition: In between sending and receiving, the receiver may become member of a different user namespace. This would invalidate the translation. The kernel should thus transmit the sender's KUID and translate it when delivering the message.

When using this approach, another problem arises: the `struct siginfo` instance contains a union. Only certain fields of this union are written to by the sender, depending on the kind of signal being sent. When the kernel delivers the signal, it is impossible to know whether the `si_uid` field is filled out. While the signal number is known, this it not enough to determine which variant of the union was written to. The `timer_create(2)` system call signals threads with a signal number determined by the thread itself. However, the signal sent by this timer does not have the `si_uid` field filled out. JITTY-OS solves this problem by transmitting the KUID externally to the `struct siginfo` instance. If a sender does not wish to send a KUID, it signifies this by setting this external field to -1. Since -1 is never a valid KUID, it may be used to encode the absence of the field. When delivering the signal, JITTY-OS checks this external field and translates the KUID into the receiver's namespace if necessary.

3.3 PID Namespaces

While user namespaces were mostly just an addition to an existing system, PID namespaces require more changes to the already existing system. The following will now present why these changes are required and how the were performed in JITTY-OS.

3.3.1 Required Changes for PID Namespaces

JITTY-OS's existing threading implementation is relatively straightforward. There is a global doubly linked list of living threads. Every thread control block contains the thread's TID, identifying the thread in the global list. With the introduction of PID namespaces, this is no longer sufficient. The same thread is not only visible in its own PID namespace, but also in every ancestor of that

PID namespace, while being assigned a different TID in each. Furthermore, the same TID may be allocated repeatedly in different PID namespaces, ruling it out as an identifier. A TID is only a valid identifier inside a single PID namespace. The kernel must therefore provide the ability to assign multiple TIDs to a single thread and to find threads within a namespace.

The structures used to manage thread groups are actually really similar to those used to manage threads. The same thought processes also apply to them. The list of thread groups and its TGIDs were previously globally unique. This changes with the introduction of PID namespaces. The kernel thus must provide more facilities to track them. In the below text, only threads and TIDs are referenced but similar changes are required for thread groups and TGIDs.

Threads should be mostly oblivious to the fact that they are part of a different PID namespace. System calls like `getppid(2)` that expose process and thread identifiers to userspace keep the usual semantic. This includes the fact that signals sent by a thread outside of a thread's PID namespace appear to be sent by a thread with identifier 0. The `kill(2)` system call should only deliver signals to threads visible within the sender's PID namespace. Similarly to user namespaces, some IPC primitives allow communication between threads in differing PID namespaces. Whenever thread identifiers are transmitted this way, as is possible using the `SCM_CREDENTIALS` ancillary message, the kernel has to be make sure the correct identifier is read by the receiver.

The first thread in a PID namespace takes a special role as its init thread. The kernel must track the init thread and treat it specially, for example with regards to signal handling. Furthermore, the kernel must prevent new threads from being created once the init thread terminates.

Furthermore, the behavior of the `proc(5)` file system changes with the introduction of PID namespaces. When the file system is mounted, the mounting thread's current PID namespace is recorded. Instead of all threads being visible in the file system, only those which are visible in that PID namespace may be listed there. The kernel has to be able to handle this change in behavior as well.

PID namespaces have relatively few dependencies. Like all other kinds of namespaces they depend on the ability to allocate memory and IDs for their structures. Like other non user namespaces, they depend on user namespaces since unsharing the PID namespace is a privileged operation in the caller's user namespace. PID namespaces and their structures need to be available for signal and process handling code. Note that this excludes the scheduler: scheduling does not behave differently under the presence of PID namespaces.

3.3.2 Structures for PID Namespaces

The structures used to implement PID namespaces in JITTY-OS are shown in Figure 3.4. PID namespaces are special regarding synchronization. Even without namespaces, the state used by JITTY-OS's scheduler is protected by a single, global lock: the `sched_lock`. This is the case because the many different variables used by the scheduler are interrelated. For example, when creating a new thread, a new TID has to be allocated. To make sure the new TID is not in use, the thread list is checked to see whether another thread with that TID exists already. In the meantime, the thread list may be modified with threads being added or removed by other CPUs. The state used to allocate TIDs and the thread list should thus be protected by the same lock.

This remains unchanged with the introduction of PID namespaces. At least in a single namespace, all this state should still be protected by the same lock. There is more leeway across different PID namespace instances. Their state is mostly unrelated. It would be possible to protect and mutate each PID namespace individually. For simplicity's sake, this is not the case in JITTY-OS. Unless marked otherwise, all variables in Figure 3.4 are still protected by the same global `sched_lock`.

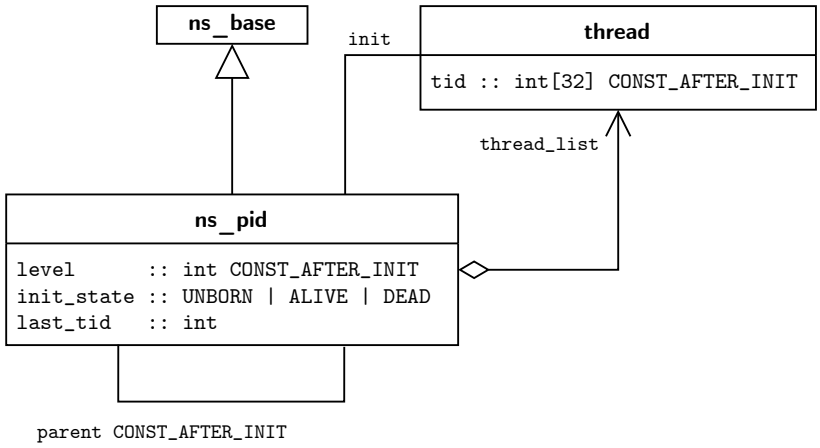

Figure 3.4 – The structures used to implement PID namespaces. The shared structure `ns_base` is described more in depth in Section 3.1.2 and Figure 3.1. Unless marked otherwise, state is protected the same as other scheduling state. In JITTY-OS. this means the state is protected by the global `sched_lock`. The `tid` array has a size of 32 since this is the maximum number of PID namespaces a thread may be visible in simultaneously.

Whenever a new PID namespace is created, the currently active PID namespace is recorded and stored in the new instance's `parent` field. This is required for two reasons: firstly, the parent of a PID namespace can be queried through the `proc(5)` file system and the `ioctl(2)` system call. Secondly: even if the parent PID namespace is not directly referenced by threads inside the child namespace, the threads still are visible inside the parent namespace. The parent namespace must thus be kept alive as there still are member threads, even if these threads are only a *direct* member of the child namespace. It would also be possible to store a reference to every PID namespace a thread is visible in on the thread instance. This is a waste of space however, since from the thread's perspective, only the PID namespace it is a direct member of is used. Since the parent of a PID namespace never changes, the field is marked `CONST_AFTER_INIT`.

To speed up the allocation of new TIDs, JITTY-OS uses the `last_tid` counter to store the last allocated TID. The allocating procedure simply increments the counter and then checks whether a thread with that TID is currently alive. Previously, there was a single global counter. The counter is now stored per PID namespace instance. The counter could in principle have stayed global. Allocated TIDs still would be unique within a single namespace. However, this would also prevent the same TID from being used more than once. This would partly defeat the purpose of PID namespaces.

3.3.2.1 Tracking multiple TIDs

JITTY-OS keeps one TID per thread per *level*. The level of a PID namespace is its number of ancestors. Storing one TID per level is sufficient as a thread will never be visible in multiple PID namespaces of a single level. PID namespaces store their level in a field during their creation to allow quick access. The value could be calculated every time it is used by walking up the ancestry chain given through the `parent` field. However, the value is used frequently and calculating it dynamically is relatively expensive. Since the number of ancestors never changes, the field is marked `CONST_AFTER_INIT`.

Threads store their TIDs in the `tid` array. This array contains a filled entry for every PID namespace the thread is visible in. This is illustrated in Figure 3.5. Linux and subsequently JITTY-OS limit the nesting of PID namespace to a maximum of 32 levels. Thus, using an array with a static

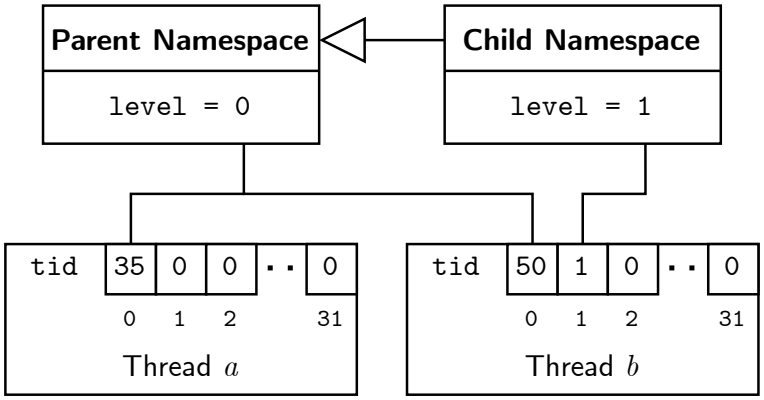

Figure 3.5 – A situation involving two threads and two PID namespaces. Thread a is only visible in the parent namespace while thread b is visible in both. The thread's `tid` arrays contain valid values for every namespace they are visible in. Other entries are initialized to 0.

length is sufficient. The actual nesting depth is expected to be low for most applications, thus most of the arrays entries remain empty. It would be possible to use a dynamic array. This approach has some drawbacks however: firstly, it would add another layer of indirection. Instead of directly storing the TIDs, they are hidden behind a pointer. Secondly, it would raise the thread struct's complexity significantly more than static array does.

To access a thread's TID at a certain level, the `tid` array is simply indexed using the relevant PID namespace's `level` property. Note that there must be special handling for cases where there is no entry for a certain level. For example, thread a in Figure 3.5 may send a signal to b since they are both visible in the parent namespace. In the signal handler, b receives the sender's TID. However, the sender is not visible to b and the TID observed by b should thus be 0. This may be implemented by checking for the sender's PID namespace level when delivering the signal to b; if it is lower than b's PID namespace level, the sender is not visible to b and the observed TID should be 0. JITTY-OS instead initializes all unused entries in the `tid` array to 0. When b accesses a's `tid` array using the level of its own PID namespace, the access will return the correct value of 0. Compared to the former approach, this avoids the complexity of branching with the drawback of having to fill unused entries.

3.3.2.2 Virtualizing the Thread List

In addition to the global thread list, JITTY-OS keeps one list per PID namespace containing every thread visible within that namespace. This allows quick retrieval of the threads in a single namespace as is necessary when sending signals for example. This is not the only possible approach. The global thread list in principle sufficient. It is possible to filter the global list and retrieve just the threads in a certain namespace. However, this has several drawbacks: it requires iterating many more threads, even if they never should be considered because they are in a unrelated namespace. Secondly, it requires the ability to check whether a thread is a member of a certain namespace. Since the thread may be a member of a child namespace, this either requires checking every ancestor of the child's namespace or storing a reference to every namespace the thread is visible in on the thread structure.

3.3.2.3 Tracking the Init Thread

JITTY-OS uses two values to keep track of a PID namespace's init thread: the `init` reference to the init thread itself and the `init_state` enumeration to keep track whether the init thread was already started and whether it has already terminated. This enumeration is necessary since the init thread is not just either alive or terminated, but may be in one of three possible states:

1. It has not been created yet. The first thread to join the namespace becomes the init thread. This is the *unborn* state.

2. It is *alive*. The PID namespace is in normal operation and the children of terminating thread groups get re-parented to the init thread's thread group.

3. The init thread has terminated. Further attempts by threads to join the namespace will fail with `ENOMEM`. This is the *dead* state.

The state could also be encoded using a number and magic values. Using an enumeration instead is more simple and easier to understand.

3.3.3 Transmitting TIDs between different PID Namespaces

As with user namespaces, JITTY-OS provides IPC mechanisms that allow communication between threads in differing PID namespaces. Likewise, the TIDs of the communication partners may be transmitted. Again, there are two cases.

Firstly, there are IPC interfaces where the sender knows which thread exactly will receive its message. With user namespaces, even if this was the case, credentials had to be translated on the receiver side. This was due to the possibility of a translation becoming invalidated while the message was being transmitted, for example by a receiver changing its user namespace. This is not the case with PID namespaces. Threads can never change their PID namespace affiliation. Sent TIDs thus may be translated on the sender site. Translating the TID only on the sender site, not on both sides as with user namespaces, simplifies the receiver side.

As an additional property, the sender's and receiver's PID namespaces are known to be related in these cases. Otherwise, the receiver would not have been visible to the sender in the first place. The receiver's PID namespace is thus either the same as, or a descendant of the sender's PID namespace. This can be exploited when sending signals: the receiver must know whether the sender is visible in the receiver's PID namespace. This is the case because PID namespace init threads handle signals differently depending on whether they were sent by threads visible or invisible in the namespace. Due to the relation between the sender's and receiver's PID namespaces, this is easy to accomplish. The transmitted TID is retrieved through the sender's `tid` array as described in Section 3.3.2.1. In case the sender is visible in the receiver's namespace, they must be in the same namespace. The translated TID then is simply the TID in that shared namespace. In case the sender is invisible in the receiver's namespace, the sender's namespace must be an ancestor to the receiver's namespace. The value looked up in the array then has the correct value of 0.

In contrast, IPC interfaces where the sender does not know which thread will be the receiver are more complicated to implement. The sender does not know which PID namespace the receiver is in and their namespaces may be completely unrelated. The TID can not be translated on the sender site and must instead be translated by the receiver. To be able to translate the TID, the receiver has to know three bits of information: every TID the sender may be associated with, which namespaces the sender is visible in and whether these namespaces are related to the receiver's own.

One way to achieve this, is to send an owning reference to the sender `thread` instance along through the IPC mechanism. However, this leads to complicated ownership tracking. It is not exactly clear when this reference is to be released. For example, if the message if never received by anyone the thread will probably never be released, even if it is practically unused. Instead, the `tid` array itself can be transmitted, alongside an owning reference to the sender's PID namespace. This would allow the receiver to iterate the ancestors of the sender's PID namespace and check whether its own namespace is related. With this scheme, the same problem lined out above arises.

This can be solved by instead sending an array of references. A reference to every PID namespace the sender is visible in is stored in this array. The elements of this array need not be dereferenced to find the ancestors. Instead, a reference to every ancestor is already part of the list. The references can thus be non-owning, removing the need to track ownership. The receiver can then check whether its namespace matches up with one of the references. If this is the case, sender and receiver are related and the TID may be translated using the process described in Section 3.3.2.1. Otherwise, the sender is guaranteed to be invisible to the sender and the calculated TID should be 0. This has higher runtime costs than transmitting a single owning reference. The list of namespaces first needs to be constructed by iterating the ancestors of the sender's PID namespace. Afterwards, the list needs to be copied over to the receiver. However, this approach is vastly more simple than tracking an owning reference and was thus chosen in JITTY-OS.

3.3.4 Adjusting the Proc File System

Displaying just the threads of a certain PID namespace after mounting an instance of the `proc(5)` file system requires keeping track of the PID namespace that was active at the time of mounting. When that namespace is known, its thread list can simply be iterated by the file system. To represent the `proc(5)` file system, JITTY-OS allocates a structure in memory representing the file system state. Keeping track of the PID namespace is then simply a matter of adding a reference to the state. The reference should be owning since the Linux semantic mandates that PID namespaces be kept alive as long as there is still a mounted `proc(5)` file system referring to them.

Additionally, the contents of files provided by the `proc(5)` file system can change depending on the PID namespace associated with the mount point. JITTY-OS handles this by passing the namespace reference through to the reading and writing procedures. This is the minimal state required to implement these procedures. The kernel could also pass around the structure representing the whole file system. However, this can easily lead to cyclical dependencies. The file system implementation already depends on the reading and writing procedures Taking this alternative approach leads to the reading and writing procedure depending on the file system as well.

3.4 Network Namespaces

The resources making up the network stack are, in principle, quite straightforward to virtualize. As with other types of namespaces, resources that were previously available as a global instance have to be part of the network namespace now. In JITTY-OS, this encompasses the list of network devices and routing information. The Linux network namespace implementation additionally virtualizes firewall rules. This is not the case for JITTY-OS since it has no firewall. Like the other kinds of namespaces, network namespaces of course depend on memory and ID allocation utilities. Additionally, since every network namespace has its own loopback network interface, they require the ability to allocate such interfaces.

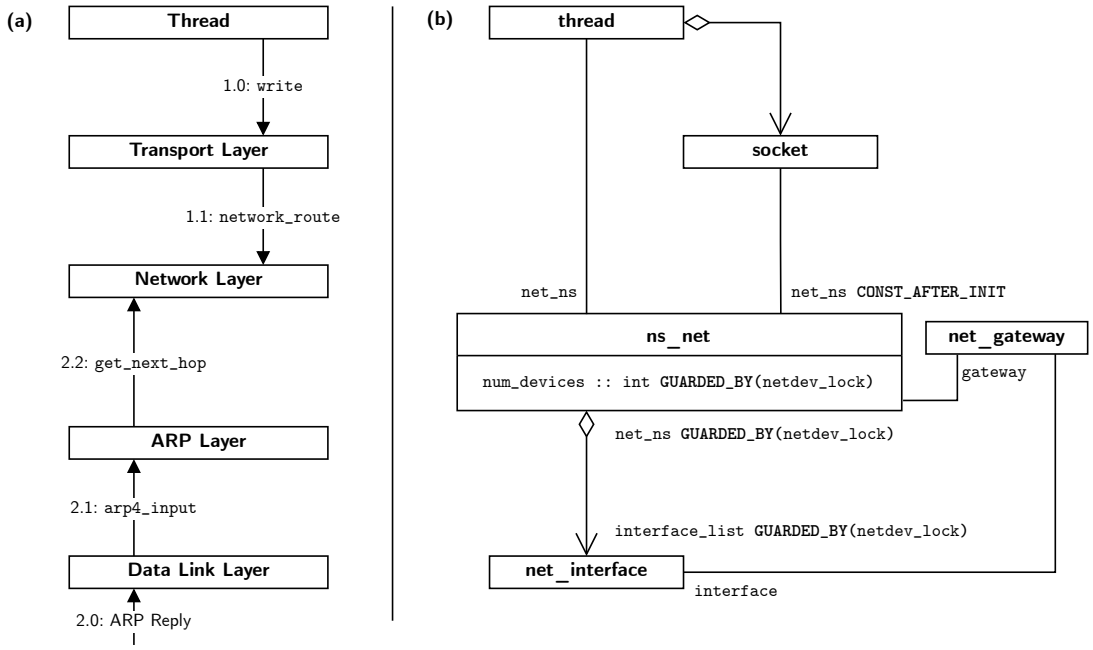

Figure 3.6 – UML diagrams describing the network stack and the integration of network namespaces

(**a**) UML communication diagram describing both output (1) and input (2) through the network stack. In the output case, a thread is sending out data. In the input case, the system is reacting to an ARP reply. Both input and output depend on the functions of the network layer for routing purposes.

(**b**) UML class diagram showing how network namespaces embed into the network stack. The `netdev_lock` is a global lock protecting the both the device lists and back-references from the interfaces to their containing network namespace. Note the parallels to **(a)**. The shared structure `ns_base` is described more in depth in Section 3.1.2 and Figure 3.1. The `net_gateway` struct is used by JITTY-OS for routing purposes.

The main difficulty in implementing network namespaces stems from the network stack's complicated dependencies. The network stack is not a simple top-down software system. Data does not only flow from higher layers to lower layers, but also in the opposite direction. Consider the two interactions illustrated by the communication diagram in Figure 3.6 (a):

1. A thread wants to output data by writing to one of its sockets (1.0). Since the communication partner may reside in a different network, but each system may only directly speak to peers in its local network, the network layer is consulted to determine a route to the target (1.1). Thus, the output process depends on the routing tables.

2. Assume an earlier process caused an ARP request to occur. Now the network card receives the reply to this request (2.0). The reply to the request is fed into the ARP layer (2.1). Due to its implementation in JITTY-OS, the ARP layer has to consult routing tables through the `get_net_hop` function in order process the reply (2.2) [Sau20]. Thus, the input process depends on the routing tables.

This specific dependency could be removed by other means. For example, the ARP layer could cache the next hop when the request is first made, rendering it unnecessary to consult the routing table when the reply is received. In the general case however, the dependency remains. For example, the system could act as a network gateway. With the primary task of a gateway being the forwarding of packets, the routing table would have to be consulted on most network inputs.

Network namespaces virtualize both routing tables and contain lists of network interfaces. Thus, they have to embed into the network stack well, allowing usage from both high and low layers. The following sections will present how this is achieved in JITTY-OS and discuss alternative solutions.

3.4.1 Embedding Network Namespaces into the Network Stack

To allow the aforementioned usage, network namespaces lie right in the middle of the network stack This is illustrated in Figure 3.6 (b). The following paragraphs will describe the used structures in depth.

Every socket holds a pointer to a network namespace. Whenever a thread establishes a connection in its current network namespace *a* by opening a socket, it should be able to switch to a different network namespace *b* while still keeping the connection intact. This allows the thread to isolate itself, prohibited from establishing new connections caused by vulnerabilities exploited through the network connection. The connection is routed over a certain network interface. That network interface is necessarily part of *a*, otherwise it would not have been possible to establish the connection in the first place. Since the socket can not rely on the thread to provide the correct network namespace, it has to keep a pointer to *a* itself. Keeping a pointer allows easy access to the network namespace when needed, for example when sending packets over the socket. Since the network namespace association will never change during the socket's lifetime, the pointer is marked `CONST_AFTER_INIT`. Because the network namespace virtualizes routing tables, the namespace contains a reference to a `net_gateway`. This is the main structure JITTY-OS uses to routing packets. If there were more structures used for routing, they would be contained as part of the network namespace as well.

The bidirectional connection between network namespace and network interface is of particular interest. The need for this pointer cycle stems directly from the dependency cycle between the network and data link layers. To send data, the correct route needs to be determined and thus a network namespace is required. When data flows from top to bottom, its easy to find this network namespace as it is part of the thread and socket. When data instead is received over one of the local network interfaces, there is no thread context. To still be able to route and forward the received packets, the network interface needs a reference to its containing network namespace.

Just like pointers to other namespaces, pointers to network namespaces are reference counted. Because of the reference cycle involving the network namespace and its contained network interfaces, special care has to be taken in this regard. It would be an easy mistake to extend the namespace's and interface's lifetime indefinitely because of a reference cycle. To prevent this, the reference from network interface to its containing namespace is weak. This is implemented through the `num_devices` counter, containing the number of devices that are part of the namespace. Whenever the reference counter is decremented, it is compared to the device counter. If all references to the network namespace originate at contained interfaces, the network namespace is released and its interfaces are migrated to the initial network namespace. It would also be possible to count the number of elements in the `interface_list` every time the value is needed. However, the interface list is implemented as a linked list for simplicity. Iterating the list every time the count is needed is expensive. Using a dedicated counter allows quick access when the value is needed, that is, when a reference to the network namespace is released.

3.4.2 Synchronizing Network Namespace Operations

Interfaces may be moved between different network namespaces. Whenever there are no strong references to a network namespace left, the network namespace is released and all its contained interfaces are moved to the initial network namespace. This can happen if there are no member threads inside the namespace and there are no other strong references. Interfaces may also be moved deliberately through the `IFLA_NET_NS_FD` operation. For example, this operation is performed by the `ip(8)` utility. This introduces new synchronization requirements. There are two main operations which require synchronization with respect to a moving network interface.

The first operation requiring synchronization is receiving input data through the interface. As lined out in Section 3.4, when data is received on an interface, the handling procedure may need access to the interface's network namespace. The access happens through the `net_interface::net_ns` back-reference stored on the interface instance. Because the back-reference is modified when the interface is moved to a different network namespace, the access must be synchronized.

Operations that move the interface run in thread context. The steps required to move an interface between two namespaces `src_ns` and `tgt_ns` are given through the `move_interface` procedure in Listing 3.4. On the other hand, network input is handled in an epilogue context. Without further synchronization, handling network input might interrupt the `move_interface` procedure at any time. The epilogue would then observe the interface to be in an invalid state. For example, the interface may be currently removed from the old namespace but not yet added to the new namespace. It is also possible for the back-reference to still reference the old namespace.

The execution of epilogues should thus be disabled while moving the interface. In JITTY-OS, acquiring a lock using `enter` automatically disables epilogues. The lock protecting the procedure thus has two tasks. It synchronizes parallel accesses and it disabled epilogues. There are three possible answers to the question, which lock to use:

1. Using a lock belonging to the network namespace
2. Using a lock belonging to the network interface
3. Using a global lock

Option 1 can be ruled out. When reading the interface's reference to its namespace, the lock must already be held. Otherwise, it is possible the namespace will be released in between the epilogue reading the reference and calling dup on the reference, leading to a use-after-free bug. If the lock were part of the namespace, the epilogue would have to read its back-reference to acquire it.

```
1  void move_interface(net_interface *iff, net_ns *src_ns, net_ns *tgt_ns)
2    THREAD_CONTEXT
3    REQUIRES(netdev_lock)
4  {
5    // Remove from old network namespace
6    src_ns->interface_list->remove(iff);
7    // Insert into new network namespace
8    tgt_ns->interface_list->insert(iff);
9    // Update back-reference
10   iff->net_ns = tgt_ns;
11 }
```

Listing 3.4 – Pseudo-code illustrating how a network interface `iff` is moved from the namespace `src_ns` to the namespace `tgt_ns`.

Using a lock stored on the interface is a valid option. However, keeping the namespace's list of interfaces requires synchronization as well, as the list may be accessed parallelly. When an interface is moved, both locks need to be held simultaneously. This has proven difficult to implement in JITTY-OS's network stack.

If the device lock were on a higher level than the list lock, the device lock would have to be acquired first. However, the list lock is sometimes held for a long time, requiring the device lock to be held for at least as long. This undermines the purpose of more finely granular locking.

Were the device lock on a lower level instead, the list lock would have to be acquired first. However, the network stack often passes around interfaces, retrieves their namespace and then operates on the namespace's list. This thus requires the interface lock to be acquired first, prior to the list lock. JITTY-OS thus uses the global `netdev_lock` for simplicity's sake.

Updating the routing tables is the second operation requiring synchronization. When an interface is moved out of a namespace, it can be necessary to update the namespace's routing tables. In case the interface is currently used in a route, that route has to be deleted since it will now become invalid. The routing table and the interface list are thus protected by the same lock. For the reasons lined out above, this is the same lock used to synchronize the back-reference from each interface to its namespace.

3.4.3 Discussing an Alternative Approach

The structures surrounding network namespaces are arguably the most complicated across the different kinds of namespaces. Cyclical pointer constructs like the cycle between network namespaces and their contained interfaces are most of the time better avoided. Care has to be taken to avoid reference cycles. Accessing the same software layer from layers above and below normally is to be avoided as well.

It is not possible to stop all data flow from the lower to the upper layers. After all, this is the purpose of many pieces of hardware: to provide data for applications to use. The network stack and its interfaces are not different in this regard. However, it is advantageous to encapsulate the interaction from bottom to top as well as possible. In the case of the network stack, the interaction from bottom to top is the receiving of packets. Each layer modifies the packet in some way and then forwards it to upper layers.

One could instead run one thread per layer. This thread would read data from the lower layers, for example using a standard `read(2)` procedure. It would then modify the read data and publish it to upper layers using a standard `write(2)` procedure. The interaction between different layers is thus encapsulated in the `read` and `write` functions and there is no direct interaction between functions of the network stack. That would also be easy to virtualize. Each namespace would have its own copy of the threads.

However, this was not explored further during the creation of this thesis, as it would go beyond its scope. Also, it is probable that the approach has other problems as well.

3.5 Mount Namespaces

Mount namespaces virtualize mount points in the VFS. The primary concern when implementing mount points, whether namespaced or not, are the structures used to represent them. The task requires linking up the structures of unrelated concrete file systems. The following sections will first present how mount points were previously implemented in JITTY-OS without the presence

of mount namespaces. Afterwards, they will show how the VFS was modified to allow virtualized mount points.

3.5.1 JITTY-OS' VFS

In JITTY-OS, the VFS is implemented using *index nodes (inodes)* and *directory entries (dentries)*. The structures are similar to those used by other VFS like the one found in Linux. An inode represents a single file or directory on a concrete file system. Inodes provide different operations. For example, a directory inode provides operations to list contained files and directories. How these operations are implemented depends on the specific concrete file system. When listing directory contents, an `ext4` file system might consult its internal data structures while a NFS might send a network request. The dentry layer lies on top of the inode layer. Each inode, either directory or file, has a corresponding dentry instance in the VFS. This is illustrated in Figure 3.7 (a).

Each concrete file system has a root inode and a corresponding root dentry. When mounting a new concrete file system into the hierarchy, the root dentry of the mounted file system is connected to the mount point dentry. This is illustrated in Figure 3.7 (b). In this relation, the mounted root is called the *upper* while the mount point is called the *lower*.

A procedure can later traverse the directory hierarchy. For example, it might be necessary to retrieve the corresponding inode for a path like `/foo` to enable creating a directory there. To resolve this, the procedure splits the path into its components `/` and `foo`. At each step, the procedure will store the dentry corresponding to these components. Once it reaches `foo`'s dentry, it detects the presence of an upper file system and traverses to the root dentry corresponding to the `btrfs` inode.

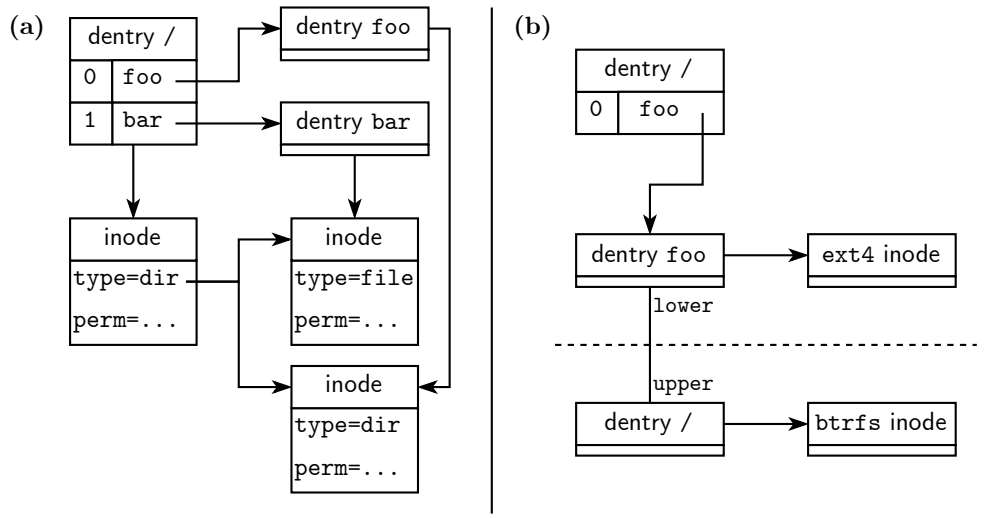

Figure 3.7 – An example of how files and folders are organized in the virtual file system (VFS).

(**a**) The `/` directory contains one file at `/bar` and an empty directory at `/foo`. Each dentry points references the corresponding inode. The inodes depend on the concrete file system and contain additional metadata, like access permissions.

(**b**) A new `btrfs` file system is mounted at the `/foo` directory. The dentries corresponding to the mount point and to the new file system's root are joined. The mount point is the lower dentry. The new root is the upper dentry.

3.5.2 Virtualizing Mount Points

The structures present in JITTY-OS as described in Section 3.5.1 only feature a singular link between upper and lower dentries. They thus only allow mounting a single upper file system at each mount point. Mount namespaces support mounting different upper file systems at the same mount point. Instead of a single link between file systems, it must be possible to store a list of links instead. When using this list, procedures traversing the directory hierarchy must make sure to only consider the link if it was created in the correct mount namespace. Furthermore, mount namespaces may be released before all their mount points are unmounted. It must thus be possible to unmount them when releasing the mount namespace's resources.

JITTY-OS achieves this using the structures illustrated in Figure 3.8. The `namespaced_mountpoint` is the implementations heart. Instead of a singular link between upper and lower file file system, each lower dentry keeps a list of mountpoint instances. Each mountpoint in turn connects to the `upper` file system. When a thread performs a mount operation, a new mountpoint instance is created and inserted into the corresponding lower dentry's upper list. On the other hand, when a thread performs a unmount operation, the mountpoint is removed from the list and released.

Every non constant piece of data is protected by the global `fs_lock`. This is not an artifact of mount namespaces. Links between dentries were already protected by this lock prior to the introduction of `fs_lock`. Since dentries remain shared both between different threads, and different mount namespace, the lock is continuously used for simplicity's and correctness' sake.

Each mountpoint holds a reference to the mount namespace it was created in. This reference can never change so it is marked `CONST_AFTER_INIT`. The mount point's lifetime is bounded by the lifetime of the namespace since releasing the namespace will lead to the mountpoint being released as well. It is thus not necessary for this reference to be owning. This reference is required to make sure only mount points in the correct mount namespace are considered when crossing mount points.

3.5.2.1 Crossing Mount Points

When traversing the directory hierarchy and crossing mount points, there are two possible directions. The procedure may traverse from lower to upper file system, and from upper to lower file system.

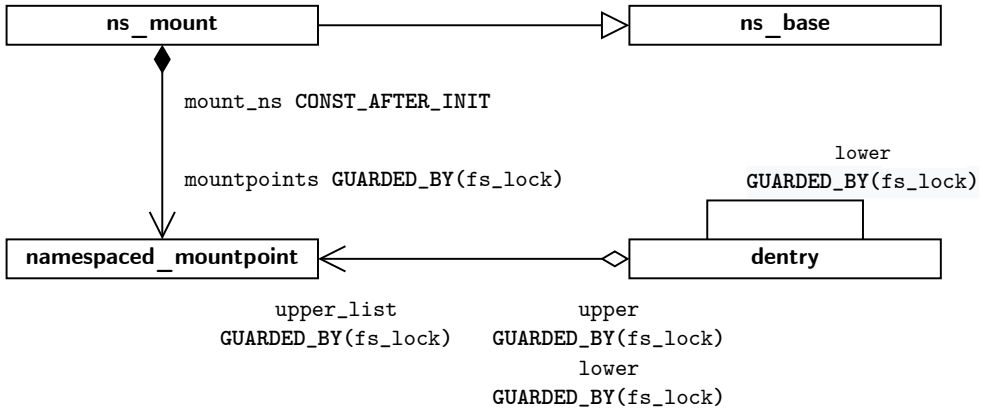

Figure 3.8 – The structures used to implement mount namespaces in JITTY-OS. The shared structure `ns_base` is described more in depth in Section 3.1.2 and Figure 3.1. The `fs_lock` is a global lock used by JITTY-OS to protect mount and unmount operations.

```
1 struct dentry *find_upper(struct dentry *lower)
2     REQUIRES(fs_lock) {
3   for (struct namespaced_mountpoint *mp : lower->upper_list) {
4     if (mp->mount_ns = ns_mount_current()) {
5       return mp->upper;
6     }
7   }
8   return NULL;
9 }
```

Listing 3.5 – The namespace aware algorithm used to cross mount points from a lower to an upper dentry

Both directions are possible using these structures. The procedure used to cross from lower to upper file system is shown in Listing 3.5. The function uses file system state and thus requires the `fs_lock`. It must iterate the current dentry's list of connected upper file systems. A link must only be considered if it is in the caller's mount namespace. When such a link is found, the corresponding upper dentry is returned.

Crossing from upper to lower dentry is easier. Each upper dentry is mounted at a maximum of one lower dentries. The upper dentry can thus link to the lower dentry directly. It would be also possible to keep a similar list from upper to lower. However, linking directly is both more simple and more efficient as it saves iterating the list.

3.5.2.2 Releasing the Mount Namespace

It is possible, that all member threads leave a mount namespace before all its mount points are unmounted. To avoid resource leaks, the mount points must be unmounted when the namespace is released. Note, that this makes mount namespaces dependent on the file system structures used to perform such operations.

In terms of data structures, this is made possible by keeping a list of mount points created in a mount namespace as part of the namespace instance. This is vastly more efficient than iterating a global list of dentries and checking whether each dentry is mounted in the namespace to be released. It also more closely models the semantic of the mount point belonging to the mount namespace.

As described in Section 3.5.2.1, each `namespaced_mountpoint` contains a reference to the upper dentry. The reference to the lower dentry is instead kept on the upper dentry. When traversing the directory hierarchy, the procedure can check whether the current dentry references any lower file system. This is not sufficient to release the mount namespace as there is no current dentry. However, the dentry must be modified to unmount the file system. This is achieved by also keeping a reference to the lower dentry on the `namespaced_mountpoint` instance.

3.5.2.3 List Implementation

While the previous sections explained why keeping lists of mount points instead of single links is necessary, they did not present how these lists are implemented. There is some leeway. The `upper_list` effectively equips every lower dentry with a mapping from namespace to upper dentry. In JITTY-OS, the mapping is implemented as an intrusive doubly linked list. As such, it requires a linear search to find the correct element. Using a hash map would allow access in constant time. However, hash maps are more complicated data structures than linked lists. For example, they must

handle resizing. In an intrusive linked list, resizing is relatively easy because links are simply stored on the list element instances. Since JITTY-OS primarily strives for simplify, a linked list was chosen over a hash map.

4 ANALYSIS

This chapter evaluates the namespace implementation presented in the previous chapter. Usually, an evaluation entails some form of performance or runtime measurement [Sau20] [Krü20] [Wic22]. This is especially reasonable when evaluating isolated components. It is easier to measure their performance as there is little interference from other components. For instance, when measuring the TCP stack's performance, a slow file system is unlikely to have any effect on the measurement.

Namespaces are not an isolated component. They only extend existing resources to support virtualization. Their introduction thus influences every part of the system that is already interrelated with the resource at hand. It is likely introducing namespaces slows down the system as a whole, as there is another layer of indirection through the namespace when accessing resources. However, it is hard to pinpoint where exactly performance is lost.

On the other hand, the wide influence of namespaces on the many subsystems has an additional possible effect: they increase the complexity of the system as a whole. An increase in complexity is best avoided. Complexity has been linked to code defects [Ebe+16]. Furthermore, one of the main goals of JITTY-OS is to be a clean and understandable Unix implementation. This chapter will thus evaluate the namespace implementation based on its complexity.

One way namespaces introduce complexity into the system is through the new structures used to manage them. For instance, this includes the life cycle functions included in every namespace's module. Their complexity is of concern for everyone that either maintains the namespaces, or that seeks to get an understanding for the system as a whole. One of the goals of the JITTY-OS project is to create an OS suitable for teaching, so the latter point is of increased importance. The rest of the chapter will be dedicated to finding metrics measuring software complexity and then apply them to the new structures.

4.1 Metrics for Software Complexity

It is hard to find an objective metric to measure software complexity. Calculating a program's cyclomatic complexity is a popular approach to measure software complexity. To calculate it, one must construct a control flow graph (CFG) for the functions making up the given program [All70]. McCabe then defines cyclomatic complexity M of the program based on the CFGs of its functions as

$$M = E - N + 2P \tag{4.1}$$

Hereby, E is the number of edges, N is the number of nodes and P is the number of connected components in the functions' CFGs [McC76]. A high number of edges in the CFG thus increases the cyclomatic complexity. This can happen for a number of reasons, for example due to excessive usage

of branching or `goto` statements. The number of connected components P in the CFG of a single function is always 1. It increases when measuring multiple functions at once, for example when considering all of the functions in a single module at once. Correlation between the cyclomatic complexity and code errors has been shown in case studies [WWM96]. Moreover, there are reports linking cyclomatic complexity to the maintainability of complex systems [Cla+08].

To calculate the cyclomatic complexity for real C code in JITTY-OS, the GCC can be instrumented with the `-fdump-tree-cfg-graph` flag [4]. This causes the compiler to emit a control flow graph for every module in `dot` format [5]. In the emitted `dot` file, nodes and edges are separated and can be easily counted using `awk` [6].

4.2 Evaluating New Structures

The following will show how the complexity of the newly introduced structures compares to the existing system. The measurements are based on commit `e15035d272d8` of the main JITTY-OS repository. Both the complexities of singular functions and of modules as a whole are compared. A higher number of functions in a module leads to higher cyclomatic complexity for that module as a whole. Showing both cases allows reasoning about the understandability of both small and larger units of the system.

JITTY-OS's modules are divided into a hardware-independent part, drivers and several hardware abstraction layers. In the analysis, only JITTY-OS's hardware-independent modules are considered. Namespaces are implemented exclusively in the hardware-independent part, so they are better comparable. Table 4.1 gives an overview over the average cyclomatic complexities of both modules and functions. The individual modules making up the namespace implementation are listed as well. The `namespace_user`, `namespace_pid`, `namespace_net` and `namespace_mount` modules contain the implementation of the respective types of namespaces. The `namespace_access` modules contains functionality used to retrieve the current thread's namespace affiliations as described in Section 3.1.2. The `namespace_fs` module contains functionality used to implement the files in `/proc/[tid]/ns`. The `namespace_syscall` module contains the implementation of the `unshare(2)` and `setns(2)` system calls.

As can be seen, the namespace implementation as a whole lowers the average cyclomatic complexity, both for the average module and for the average function individually. McCabe classifies individual functions with cyclomatic complexity values of up to ten as simple with little reliability risk. A value between 11 and 20 signifies moderate risk, a value between 21 and 50 signifies high risk [McC08]. On average, the namespace functions lay below a value of 10. The functions of the `namespace_syscall` module are an exception to this. This is due to the fact that the `unshare(2)` and `setns(2)` system calls branch heavily; both support many different operations. It should be noted, that while the namespace's complexity values are generally free of outliers, the `create_new_mapping` function of the `namespace_user` module has a relatively high complexity value of 63. This is caused by the function handling the creation of both UID and GID translation tables. Furthermore, the function has a lot of error conditions.

The average function complexity is not necessarily correlated with the module's complexity. The function complexity values are added up to form the module complexity value. Furthermore, the number of functions is factored in a second time through the P factor. A module thus quickly becomes rated as more complex if it contains many functions. The `namespace_user` module has

[4] https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Developer-Options.html#index-fdump-tree
[5] https://graphviz.org/doc/info/lang.html
[6] https://www.gnu.org/software/gawk/manual/gawk.html

a higher complexity than the other namespace modules. This again is due to the need to read in translation tables.

Table 4.1 – The average cyclomatic complexities of preexisting JITTY-OS modules and the namespace implementation.

Module	**Avg. Cyclomatic Complexity**	
	Per Function	Per Module
Hardware Independent	10.37	135.67
Hardware Independent except Namespaces	10.53	137.82
Namespaces Average	8.31	88.57
Namespace modules individually		
`namespace_syscall`	25.75	103
`namespace_pid`	8.46	110
`namespace_user`	7.04	183
`namespace_net`	6.56	105
`namespace_mount`	4.67	28
`namespace_fs`	3.68	81
`namespace_access`	2.00	10

4.3 Support for systemd

As presented in Chapter 1, one of the goals of this thesis was to support systemd as shipped in Debian starting at Debian 7.0. However, booting and running new Debian versions requires additional kernel features apart from namespaces which are currently unsupported in JITTY-OS. Implementing the other features is an ongoing effort. This means however, that support for systemd was not evaluated at the time of writing.

4.4 Conclusion

As could be seen in the previous section, the namespace implementation does not significantly increase the system's complexity, relative to JITTY-OS's previously implemented modules. It should be noted however, that some singular functions still present a high complexity value. The `create_new_mapping` function used to read in new translation tables for user namespaces is particularly complex. It can also be concluded, that on average, functions in JITTY-OS are limited in complexity. This is in accordance with the goal of remaining simple. Whether the namespace implementation supports new versions of Debian and systemd remains to be tested.

5 CONCLUSION

This thesis successfully implements user-, PID-, network- and mount namespaces without mount propagation in JITTY-OS. This lays the foundation for further support of other kinds of namespaces, and support for OS-level virtualization software in general, such as Docker and systemd. Moreover, the ability to test programs in isolation in a namespace is useful for debugging. Chapter 2 introduced the basics of OS-level virtualization. Based on this, Chapter 3 presents a framework for any kind of namespace based virtualization, followed by the concerns of the specific namespace types. In turn, Chapter 4 seeked to evaluate the structures presented before in their simplicity.

The implementation was evaluated in its simplicity. Compared to JITTY-OS's existing modules, the new implementation does not increase the average module or function complexity. It remains to be tested whether new versions of Debian and systemd are supported by the implementation. This is a currently ongoing effort.

There still remains much work to be completed in the future. Apart from the namespaces presented in this thesis, the Linux kernel additionally supports namespaces around cgroups, IPC primitives, the system time, host names and domain names. These could be implemented in JITTY-OS as well. Due to the shared structures introduced in Section 3.1, supporting new kinds of namespaces should come at a manageable cost. Note, that JITTY-OS currently does not support cgroups or most of the IPC primitives virtualized by IPC namespaces. In order for such a namespace implementation to be useful, the virtualized resources must first be implemented themselves.

Support for OS-level virtualization software like Docker can be explored further. This requires additional runtime support. Among other features, this includes an overlay file system, the secure computing mode (seccomp) and virtual network devices. Furthermore, due to the work done by Fischer, JITTY-OS allows thread to chose between the Linux and the FreeBSD ABIs at runtime [Fis22]. FreeBSD supports jails, an OS-level virtualization framework comparable to namespaces. One could investigate how compatible the two mechanisms are. This allows one to potentially execute containers of different operating systems using a single kernel.

The namespace implementation introduced by this thesis supports multithreading through the use of locks. It uses the simple locking constructs provided by JITTY-OS. Performance may be increased by implementing compile-time lock optimization. This includes the credential translation tables presented in Section 3.2 in particular as they are a candidate for a readers-writer lock.

LIST OF ACRONYMS

ABI	application binary interface
API	application programming interface
ARP	address resolution protocol
CFG	control flow graph
cgroup	control group
CPU	central processing unit
dentry	directory entry
FAU	Friedrich-Alexander-Universität Erlangen-Nürnberg
GID	group ID
HDD	hard disk drive
inode	index node
IO	input/output
IPC	inter-process communication
IP	internet protocol
ISA	instruction set architecture
JIT	just in time
KGID	kernel group ID
KUID	kernel user ID
MAC	media access control
NFS	network file system
OS	operating system
PID	process ID
seccomp	secure computing mode

SSD	solid state drive
s-lock	spinning lock
TCP	transmission control protocol
TGID	thread group ID
TID	thread ID
UID	user ID
VFS	virtual file system
w-lock	waiting lock

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ALGORITHMS

REFERENCES

[All70] Frances E. Allen. “Control Flow Analysis.” In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: https://doi.org/10.1145/390013.808479.

[Ber14] David Bernstein. “Containers and Cloud: From LXC to Docker to Kubernetes.” In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84. DOI: 10.1109/MCC.2014.51.

[Bou16] Jonathan Boulle. “The Open Container Initiative: Establishing Standards for an Open Ecosystem.” In: *Open Source Forum*. The Linux Foundation. Nov. 15, 2016. URL: http://events17.linuxfoundation.org/events/archive/2016/open-source-forum (visited on 08/23/2022).

[Cla+08] M. N. Clark et al. “Measuring software complexity to target risky modules in autonomous vehicle systems.” In: *AUVSI North America Conference* (2008). URL: http://www.mccabe.com/pdf/MeasuringSoftwareComplexityUAV.pdf (visited on 08/27/2022).

[DER22] Trevor Dunlap, William Enck, and Bradley Reaves. “A Study of Application Sandbox Policies in Linux.” In: *Proceedings of the 27th ACM on Symposium on Access Control Models and Technologies*. SACMAT ’22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 19–30. ISBN: 9781450393577. DOI: 10.1145/3532105.3535016. URL: https://doi.org/10.1145/3532105.3535016.

[Ebe+16] Christof Ebert et al. “Cyclomatic Complexity.” In: *IEEE Software* 33.6 (2016), pp. 27–29. DOI: 10.1109/MS.2016.147.

[Fis22] Nicolai Fischer. “Extending the Linux-Compatible JITTY-Kernel to Support the FreeBSD ABI at Runtime.” Bachelor’s Thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, Aug. 2022.

[HBS14] DeLesley Hutchins, Aaron Ballman, and Dean Sutherland. “C/C++ Thread Safety Analysis.” In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 2014, pp. 41–46. DOI: 10.1109/SCAM.2014.34.

[Hei22] Tobias Heineken. “Entwicklung einer compilergestützten Thread-lokalen statischen Verifikation von Funktionen bezüglich ausgewählter Eigenschaften für ein Betriebssystem in C.” Master’s Thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, Aug. 2022.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Revised First Edition. Morgan Kaufmann, 2012. ISBN: 978-0-123-97337-5.

[Ker15] Michael Kerrisk. *user_namespaces(7)*. Linux man-pages project. Version Commit ecb0ff30e86e. Mar. 4, 2015. URL: https://git.kernel.org/pub/scm/docs/man-pages/man-pages.

[Krü20] Fabian Krüger. "Design und Implementierung der TCP-Schicht im JITty Betriebssystem mit minimalem und einfachem Locking." Bachelor's Thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, Dec. 2020.

[KW00] Poul-Henning Kamp and Robert NM Watson. "Jails: Confining the omnipotent root." In: *Proceedings of the 2nd International SANE Conference*. Vol. 43. 2000, p. 116.

[Lar20] Michael Larabel. *The Linux Kernel Enters 2020 At 27.8 Million Lines In Git But With Less Developers For 2019*. Jan. 1, 2020. URL: https://www.phoronix.com/news/Linux-Git-Stats-EOY2019 (visited on 08/29/2022).

[McC08] T.J. McCabe. *Software Quality Metrics to Identify Risk*. Department of Homeland Security - Software Assurance Working Group. Jan. 31, 2008. URL: http://www.mccabe.com/ppt/SoftwareQualityMetricsToIdentifyRisk.ppt (visited on 03/29/2022).

[McC76] T.J. McCabe. "A Complexity Measure." In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[Pet15] Borislav Petkov. *arch/x86/lib/memcpy_64.S Line 32*. Version Tag v5.15/Commit 8bb7eca972ad. Feb. 4, 2015. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git.

[Ros16] Rami Rosen. "Namespaces and cgroups, the basis of Linux containers." In: *NetDev 1.1*. NetDev Society. Feb. 10, 2016. URL: https://legacy.netdevconf.info/1.1/proceedings/slides/rosen-namespaces-cgroups-lxc.pdf (visited on 08/23/2022).

[Sal66] Jerome Howard Saltzer. "Traffic control in a multiplexed computer system." PhD thesis. Massachusetts Institute of Technology, 1966.

[Sau20] David Sauerwein. "Design and Implementation of a Simple Multithreaded UDP/IP Network Stack Including an E1000 Network Adapter Driver for a POSIX OS." Bachelor's Thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, Apr. 2020.

[Sie+20] Volkmar Sieh et al. "JITTY — ein einfaches, portables, Linux-kompatibles Betriebssystem." In: *Herbsttreffen 2020*. Gesellschaft für Informatik - Fachgruppe Betriebssysteme. Aachen, Sept. 25, 2020. URL: https://web.archive.org/web/20210518201539/https://www.betriebssysteme.org/wp-content/uploads/2020/09/Sieh_FAU.pdf (visited on 05/18/2021).

[Sol+07] Stephen Soltesz et al. "Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors." In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: Association for Computing Machinery, 2007, pp. 275–287. ISBN: 9781595936363. DOI: 10.1145/1272996.1273025. URL: https://doi.org/10.1145/1272996.1273025.

[Sta17] Misty Stanley-Jones. *Docker Documentation - Use bind mounts*. Aug. 10, 2017. URL: https://web.archive.org/web/20220819020306/https://docs.docker.com/storage/bind-mounts/#configure-bind-propagation (visited on 08/19/2022).

[Tan15] Andrew S. Tanenbaum. *Modern Operating Systems*. Fourth Edition (Global Edition). Pearson, 2015. ISBN: 978-1-292-06142-9.

[TC04] Andrew Tucker and David Comay. “Solaris Zones: Operating System Support for Server Consolidation.” In: *Virtual Machine Research and Technology Symposium*. 2004.

[Wic22] Markus Wich. “Design and Implementation of a 64-Bit RISC-V Port with Multicore Support for the Unix-like JITTY-OS.” Master’s Thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, Apr. 2022.

[WWM96] Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. *Structured testing: A testing methodology using the cyclomatic complexity metric*. Vol. 500. 235. US Department of Commerce, Technology Administration, National Institute of . . ., 1996.