Floating Processes across Namespaces

Jaya Gupta and Harshit Bansal
Supervisor: Prof. Debadatta Mishra

November 27, 2023

Abstract

Support for different namespaces in the Linux kernel is one of the core enablers for container execution platforms such as Docker. This project aims to design OS-level primitives and provide user-friendly interfaces to migrate running processes across namespaces. Apart from enabling the movement of processes across process hierarchies with different namespaces, the proposed design provides support for quick migration inside the same host with negligible memory overheads. The proposed feature can potentially support efficient application consolidation and chained execution with state forwarding in FaaS computing platforms.

I Introduction

Modern computer applications today have a large number of dependencies on pre-installed applications, shared libraries, and system settings. As a result of the complex interdependent nature of many applications, some applications cannot coexist on a single system with other applications. To solve these problems, it has been known to "virtualize" applications from one another and from the system software, i.e. the operating system. This is where the containers sweep in. Lightweight virtualization (i.e., containers) offers a virtual host environment for applications without the need for a separate kernel, enabling better resource utilization and improved efficiency and allowing for enhanced security and policy reinforcement. Support for different namespaces in the Linux kernel is one of the core enablers for container execution platforms such as Docker. However, application virtualization requires files and packages on which the application depends to be present within the virtualized applications. Therefore, if a program requires the JAVA runtime library to operate, the entire JAVA runtime library must be present in the virtualized system. This can lead to large size of containers. Consider the following example in Figure 1[1].

Virtualized application packages A and A′ are executed in their respective containers B and B′. Let's say, A after certain execution requires some packages that are installed in B′. Since the virtualization layer isolates 2 different containers, this could only be possible if the required package is also installed in container B. This, as described above, can lead to the bloated size of containers.[1] There is one simple solution to this, as done by many others. It is to dump the state of application A in some file and then restore that state when launching it again in container B′. However, this design is very slow and memory-intensive.

To overcome this restriction, this project aims to design OS-level primitives and provide user-friendly interfaces to migrate running processes across namespaces. Apart from enabling the movement of processes across process hierarchies with different namespaces, the proposed design provides support for quick migration inside the same host with negligible memory overheads. Current Linux kernel utilities like `setns`, `unshare`, etc. do not provide any way to change the PID namespace of an executing process. Moreover, it prevents the process that has dropped capabilities from regaining those capabilities and hence prevents movement from a lower privilege namespace to a higher privilege one. We will also explore a few use cases of this utility.

II Background

In this section, we first describe the namespace concept in the Linux kernel and how it is adopted by containers. We will then discuss various namespace API's available and their limitations.

II.I Namespaces and containers [2]

Namespaces provide an isolated virtual view of a global resource to the processes in that namespace. It is similar to how a virtual view of hardware resources is presented to processes in virtual memory etc. As per the Linux man page:

A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace but are invisible to other processes. One use of namespaces is to implement containers.

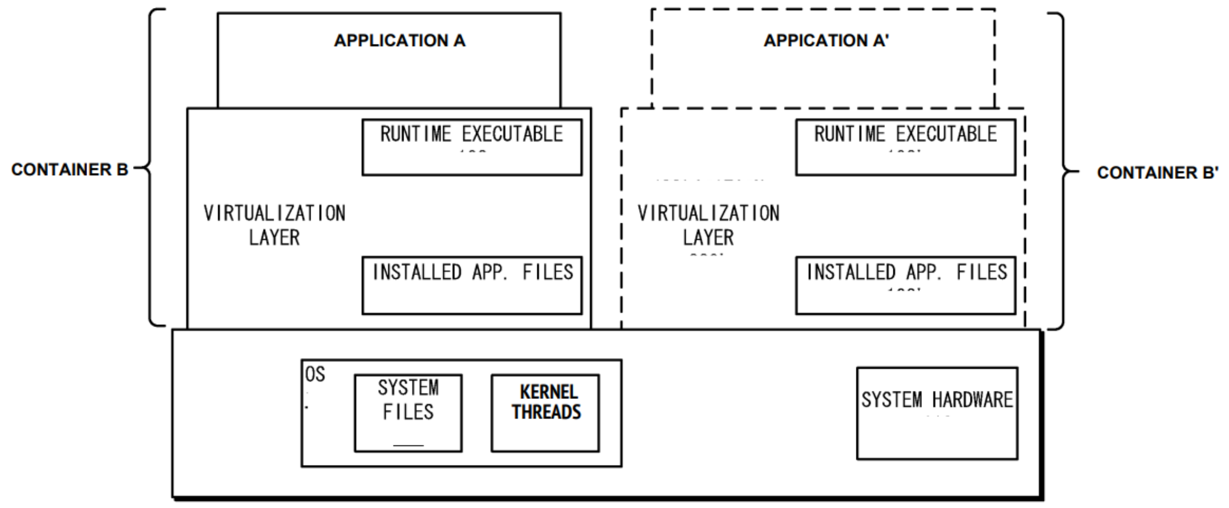

Figure 1

We use mount namespace as an example. Without mount namespace enabled, processes running within a Linux OS share the same filesystems. Any change to the filesystems made by one process is visible to the others. To provide filesystem isolation across processes, `chroot` [3] was first introduced but then found to be vulnerable to a number of attacks [4, 5]. As a more principled approach, Linux kernel introduced the mount namespace abstraction to isolate mount points that can be seen by the processes. A mount namespace restricts the filesystem view to a process by creating separate copies of vfs mount points. Thus, processes running in different mount namespaces could only operate over their own mount points. To date, seven namespace abstractions have been introduced into the Linux kernel and four namespace API's are provided to user-mode for namespace manipulation as shown in Figure 2[6].

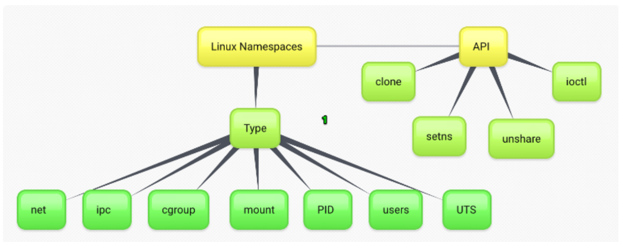

Figure 2

The functionalities of the different namespaces are discussed as follows:

- `IPC (CLONE_NEWIPC)`: IPC namespaces isolate certain IPC resources, namely, System V IPC objects like semaphores, shared memory segments and POSIX message queues.

- `Network (CLONE_NEWNET)`: Network namespaces provide isolation of the system resources associated with networking: network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, port numbers (sockets), and so on.

- `Mount (CLONE_NEWNS)`: Mount namespaces provide isolation of the list of mount points seen by the processes in each namespace instance. Thus, the processes in each of the mount namespace instances will see distinct single-directory hierarchies.

- `PID (CLONE_NEWPID)`: PID namespaces isolate the process ID number space, meaning that processes in different PID namespaces can have the same PID. Some of the notable features of PID namespace are as follows:

  - First process created in a new PID namespace has PID 1, and is the "init" process for the namespace.
  - PID namespaces are nested: each PID namespace has a parent, except for the "root" PID namespace. See Figure 3[7].
  - The parent of a PID namespace is the PID namespace of the process that created the namespace using namespace API's.
  - A process can see processes and hence spawn a child in its own PID namespace and in the descendant's namespace.

- `User (CLONE_NEWUSER)`: User namespaces iso-

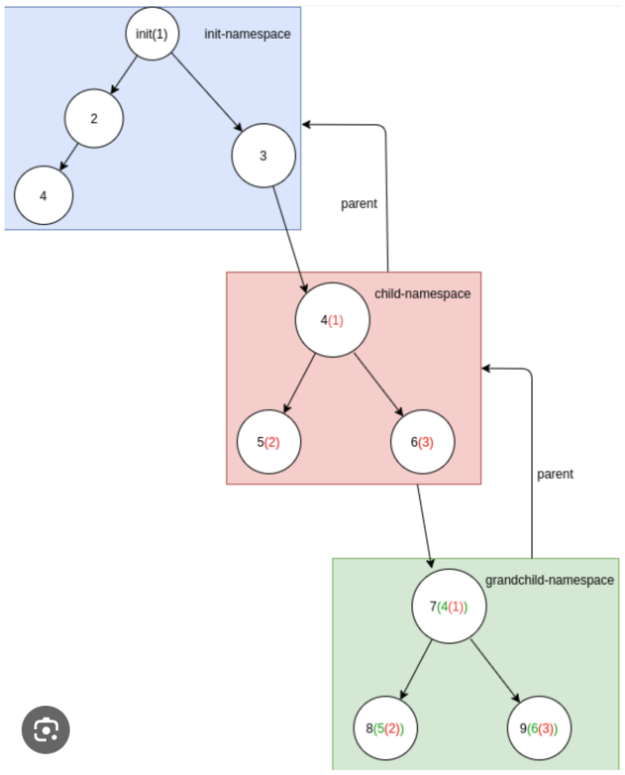

Figure 3: *Hierarchial structure of PID namespace*

lates security-related identifiers and attributes, in particular, user IDs and group IDs, the root directory, keys, and capabilities.

- `UTS (CLONE_NEWUTS)`: UTS namespaces provide isolation of two system identifiers: the hostname and the NIS domain name.
- `CGroup (CLONE_NEWCGROUP)`: Cgroup namespaces virtualize the view of a process's cgroups.

All namespaces are summarized in Table1. How services like Docker uses namespaces for isolation and container creation is shown in Figure 4. Read man page for more details.

II.II Namespace API's

Linux Kernel provides 4 APIs to change a process's namespaces. However, each API has its own limitation as we will discuss further.

- `unshare`: The `unshare` system call allows a process to disassociate from specific namespaces, creating a new namespace. Calling unshare with `CLONE_NEWPID` flag changes the PID namespace of subsequently created child.
- `setns`: The `setns` system call allows a process to join an existing namespace, effectively associating itself with the namespace of another process. Calling setns with `CLONE_NEWPID` flag changes the PID namespace of subsequently created child.
- `clone`: The `clone` system call is a versatile mechanism for creating new processes, enabling the caller to share or unshare various namespaces with the newly created child.
- `ioctl`: The `ioctl` system call is primarily employed for managing and querying properties of existing namespaces.

Table 1: Namespaces in Linux kernel.

Namespace	Constant	Isolates
IPC	CLONE_NEWIPC	System V IPC, POSIX message queues
Network	CLONE_NEWNET	Network devices, stacks, ports, etc.
Mount	CLONE_NEWNS	Mount points
PID	CLONE_NEWPID	Process IDs
User	CLONE_NEWUSER	User and group IDs
UTS	CLONE_NEWUTS	Hostname and NIS domain name

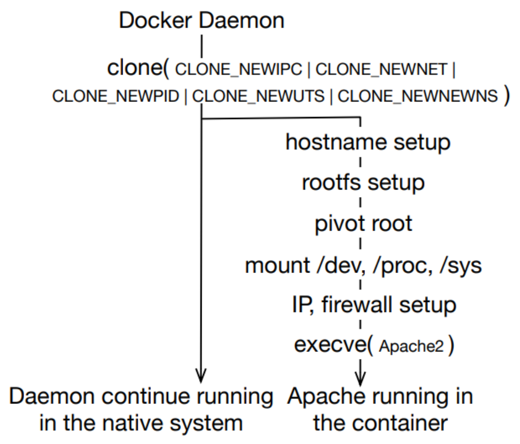
Docker Daemon
clone(CLONE_NEWIPC | CLONE_NEWNET | CLONE_NEWPID | CLONE_NEWUTS | CLONE_NEWNEWNS)
hostname setup
rootfs setup
pivot root
mount /dev, /proc, /sys
IP, firewall setup
execve(Apache2)
Daemon continue running in the native system
Apache running in the container

Figure 4: *Creating a Docker container [2]*

II.III Limitations of APIs

- `setns` requires a file descriptor of the target namespace and thus is useless in the absence of any namespace.
- `setns` and `unshare` can change all namespaces of the currently executing process except the PID namespace, as shown in the table. Hence, a process PID namespace has been fixed since its birth.
- It is not permitted to use `setns` to enter any ancestor's namespace. Hence, the movement in the namespace hierarchy is only allowed in one direction. A process can move downward from a higher privilege namespace to a lower privilege one or migration is only allowed to the

descendant's namespace.

Changes in Namespace of currently executing process		
Namespace API	**unshare**	**setns**
Change Network Namespace	✓	✓
Change PID Namespace	✗	✗
Change Mount Namespace	✓	✓
Change UTS Namespace	✓	✓
Change User Namespace	✓	✓
Change CGroup Namespace	✓	✓
Change IPC Namespace	✓	✓

Hence, in this project, we will propose a design to enable the movement of processes across namespace hierarchies. This will allow migrating the process from a higher privilege namespace to a lower privilege and vice-versa. We will majorly focus on the migration of executing process across different pid namespaces which is not yet supported by Linux.

III Motivation

In this section, we will list some use cases of this utility and the fundamental challenges of achieving it.

III.I Usage Scenarios

Certain use cases are explained below.

Chained Migration for Computation: The concept of chained migration emerges as a powerful tool in scenarios where a computational task necessitates assets from different namespaces. By orchestrating the seamless migration of a process across namespaces, we can efficiently harness resources distributed across diverse environments, ensuring a cohesive and uninterrupted computation. The ability to carry forward the state of a process during dynamic migration is the cherry on top. We can leverage this capability to maintain the integrity of ongoing computations, simplifying the complex task of preserving and transitioning the state across various namespaces.

Enhanced Security through Gradual Privilege Elevation: This project underscores a security-centric use case wherein a process is initiated in a restricted namespace and dynamically migrates to a more privileged one after successfully passing security checks. This methodology not only bolsters security measures but also ensures a controlled and secure execution environment.

III.II Challenges

Some of the challenges of dynamic migration are highlighted below.

1. **Consistency and Coherence:** Ensuring consistency and coherence across namespaces during migration is challenging. The design must account for the potential for conflicts or inconsistencies in resource states between the source and destination namespaces. Consider the PID of a process as an example. It could happen that the migrating process uses its PID for communication etc. How to ensure that the migrating process gets the same PID in the new namespace? Also, what to do if the existing PID is not available? This leads to the problem of **Resource Constraint**.

2. **Security Concerns:** Security is a paramount concern when migrating processes across namespaces. Unauthorized access or unintended privilege escalation must be prevented. Effective mechanisms for authentication and authorization need to be implemented to safeguard against potential security breaches.

3. **Resource Cleanup and Leakage:** Managing resource cleanup and preventing resource leakage is a challenge. When a process migrates from one namespace to another, resources associated with the original namespace need to be appropriately released to avoid resource contention and potential leaks.

IV Design Overview

The proposed design focuses on the seamless migration of a process from the source PID namespace (NS_1) to the target PID namespace (NS_2). This migration is achieved through a carefully orchestrated series of steps that involve detaching the process from its current parent and subsequently attaching it to a user-specified target process within the destination PID namespace.

IV.I Detachment from Parent Process

4.1.1 Descheduling Mechanism

Prior to migration, the process undergoes a descheduling phase, wherein its execution is temporarily halted. This ensures that the process is in a quiescent state, minimizing the risk of inconsistencies during the migration process.

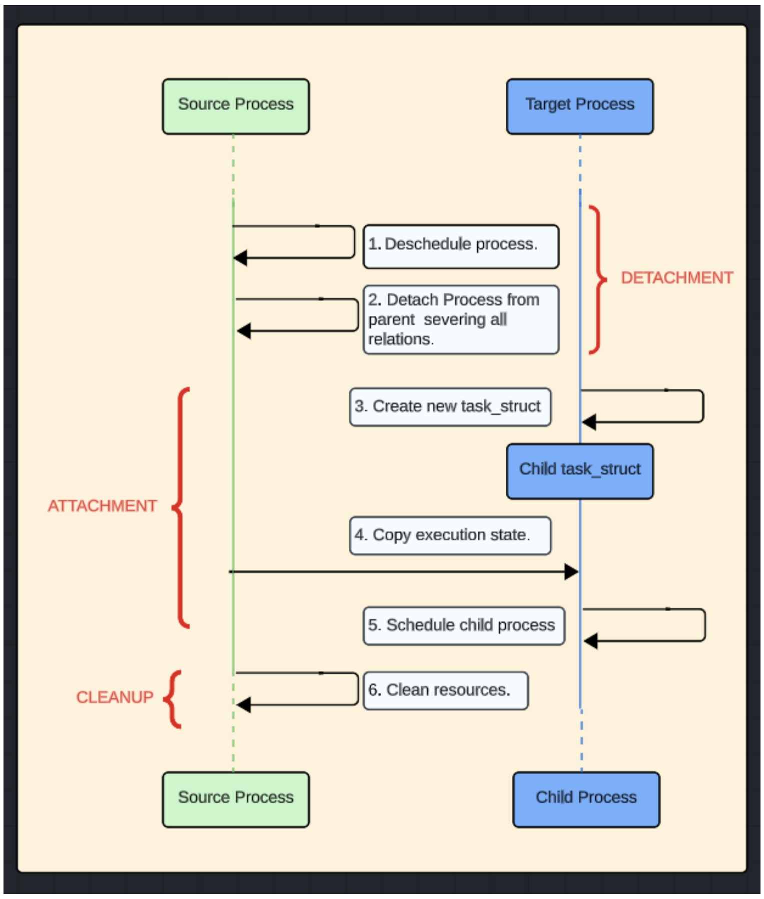

Figure 5: *Steps involved in migration*

4.1.2 Severing Parental Relationships

During the descheduling phase, all existing parent-child relationships are severed to ensure a clean break from the current PID namespace (NS1). This involves disassociating the migrating process from its parent, removing the process from the thread group, and many more things which are discussed in later sections. The severance of parental relationships is a critical step to guarantee the independence of the migrating process.

IV.II Attachment to Target Process in NS_2

4.2.1 Establishing Relationships in NS_2

Once detached from the original parent process and the source PID namespace (NS_1), the migrating process is then attached to the user-specified target process within the destination PID namespace (NS_2). This attachment involves establishing new parent-child relationships and ensuring that the migrating process seamlessly integrates into its new execution context. The old execution state(memory and register states) is copied to the new task_struct and the attached process resumes its execution.

V Implementation Details

The whole utility is structured into two components: the userspace library and the kernel module. The userspace library exposes high-level functions to client programs for migrating a process to a target parent. The library, after some processing, then communicates the request to the kernel module via Chardev. The kernel module internally migrates the process in two major phases: detaching the process from its current parent in the first phase and then attaching it to the target parent in the second phase. To accomplish them, a few changes were made in the existing system calls of `clone`, `wait`, and `exit`.

The entire codebase can be found `here`.

V.I Kernel Module

The kernel module receives the migration request through chardev. It validates the source pid (pid of the process to migrate) and the target pid (pid of the parent to migrate to) passed by the userspace library. Next, the `task_struct` of the migrating process is retrieved using kernel function `pid_task`, and the detach function is invoked.

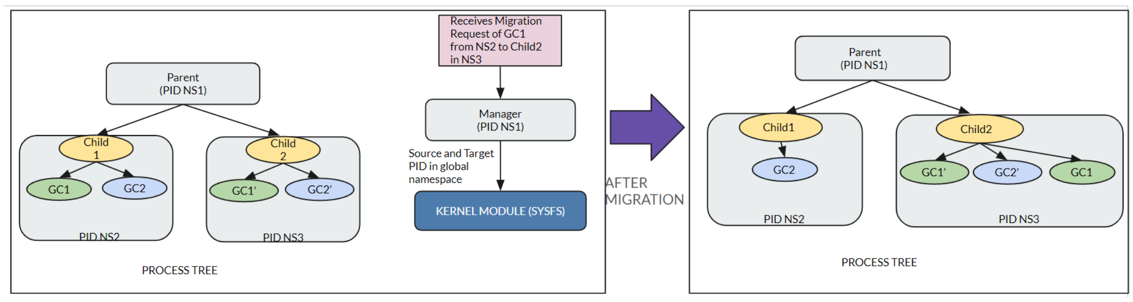

Figure 6

Principles of Detach: The detach function performs a series of operations, but before mentioning them it is important to understand why those operations are needed.

- The migrating process cannot be allowed to run in the middle of migration simply because if it makes any system call it may or may not have any parent and data structures might be in a messed up state. Another reason we cannot allow this is that we need it in a quiescent state, minimizing the risk of inconsistencies.

- Next, the current parent of the migrating process has to be fooled into believing that its child has actually called `exit`. To do this, we need to do what an exiting process does to notify its parent of its exit, and it turns out to be incredibly simple. Just send `SIGCHLD` signal to parent.

- Finally, we need to sever all links of the migrating process with its parent, and normally it is done by the parent in the wait system call. But parent also destroys all the resources of the child process by decrementing their reference count. Since obviously we don't want that, we introduce a new exit state `EXIT_MIGRATE` in the Linux kernel and handle it, especially inside the wait call. So when the parent calls wait after it has received SIGCHLD from the migrating process, it will treat it as a zombie process and remove all its references from data structures like children list, group, task list, etc.

- The above approach of severing the links assumes that the parent will call and wait in the near future, but that might not be the case. So to handle that scenario, we check if the parent is already inside the wait call, if yes, then we continue with the above approach, else we manually remove the references of a child from the parent without notifying the parent of the child's exit.

To summarize the above long points, first, the migrating process is stopped and disallowed from running during migration, next if a parent is inside the wait call it is notified of the migrating child exit and the parent severs the links with the child; else we manually do the same but without informing the parent that its child has exited.

Working: The previous section explains the main principles guiding the design of the detach mechanism, this section will explain how these principles are put into effect inside Linux kernel.

1. The migrating process is stopped by sending `SIGSTOP` signal to it, which changes the state of the process to `TASK_STOPPED`. Using `SIGSTOP` comes with its advantage that the process can now only be scheduled again on receiving a `SIGCONT` signal, all other signals have no effect.

2. To notify the parent of child exit, we just use `send_sig` function of the kernel to send `SIGCHLD` to the parent.

3. Whether we need to notify the parent of the exit of the migrating process or not depends on whether a parent is inside the `wait` system call. Whenever a parent calls wait, it first enqueues itself into its `wait_queue`. So we just check the entries of the wait queue of the parent process and if it is nonempty, it means the parent is inside the wait call and we send the signal. Otherwise, we execute the same functions inside the kernel module as done by wait call to remove the references of a child, without notifying the parent. Now there can be a potential race between checking the wait queue and parent enqueuing itself in it. So we enclose both the code sections inside a spinlock.

At this point, the migrating process is completely detached from its current parent, or we can now say 'old parent', and is ready to meet its new parent!

V.II Attach Mechanism

On reaching the attach phase of migration, we have the detached `task_struct` of the migrating process.

The attach phase has the responsibility to make the migrating process a child of the target parent and resume the execution of this new child right from the point where it stopped.

Principles of Attach: It is easy to observe that to attach a process to a new parent process we can just do the reverse of detach, i.e. add references to a new child in the parent's data structures. But as it turns out there is a much easier and better way to perform attach than this. Here we build up the mechanism in steps:

- First observe that we just care about resuming the execution of the process under the new parent, and the only data structures that are required to achieve this are `mm_struct` and registers. Note that we are concerning ourselves only with the memory state for now.

- So if we change the `mm_struct` and registers of a process with those of migrating process, then that process would basically be running the same program what migrating process was running. And changing the `mm_struct` and registers is very easy and straightforward. To meet our requirements, we need the process whose structs we change is actually a child of our target parent. So we create one such child using textcolororange`clone` system call!

- All seems fine but there is a problem with using `clone`: it executes in the context of `'current'` process that is running on cpu. So we cannot call clone from any other process but the target parent. That means we need to make the target parent call clone system call by gaining control of its execution. This is done using `ptrace` system call.

- After the parent calls clone, it will create a new process in its namespace and copy its execution state (mm_struct, registers) into the child, and insert it into the ready queue. Before inserting child into the ready queue, we change the above-mentioned data structures to those of a detached process. To achieve this special behavior in clone, a new flag `CLONE_ATTACH` is created and a clone call is made with this flag.

- Since we discard the `mm_struct` of newly created child anyway, there is no point in allocating it, so to avoid that we use `CLONE_VFORK` flag too. The parent process is not put to sleep (default behavior of `vfork`) if the `CLONE_ATTACH` flag is also set.

VI Experimental Setup

The experimental setup is designed to assess the effectiveness of the proposed namespace migration mechanism, focusing on chained execution in different namespaces with state forwarding. The setup comprises a hierarchical structure of processes across multiple PID namespaces, with migration initiated by a child process in response to a computation trigger. There is a manager service running in the global namespace that accepts socket connection and receives migration requests. The setup is shown in Figure 7. The following components constitute the experimental environment:

VI.I Process Hierarchy and PID Namespace Creation

- Parent Process (NS_1): Initiates the experimental setup and spawns four child processes in four distinct PID namespaces (NS_2, NS_3, NS_4, NS_5).

- Child1 (NS_2): Spawns 2 child processes. C1_A is the process that will do chained execution.

- Child2 (NS_3), Child3 (NS_4), Child4 (NS_5): Spawns 3, 4, and 5 children respectively.

VI.II Migration Trigger and Manager Mechanism

- **Computation-Triggered Migration (NS_2 to NS_3 to NS_4 to NS_5):** One child process under NS2 performs a computation (increment a variable 100 times in a loop) and sends a migration request to the manager process running a socket in the global namespace. This sets in motion the chained migration process across different PID namespaces.

- **Manager Process:** Receives migration requests and triggers the migration process by making a `__migrate` call to the library, which further invokes the kernel module responsible for performing the migration.

VI.III Observations and Demonstrations

The child process in NS_2 prints the computed value (initially 100) and its PID(3) before initiating migration to NS_3. After migration, the process repeats the computation twice, printing the computed value(200) and its PID(5) in the new namespace (NS_3). This process is repeated for NS_4(300, 6) and NS_5(400, 7), demonstrating chained execution with state forwarding across multiple namespaces.

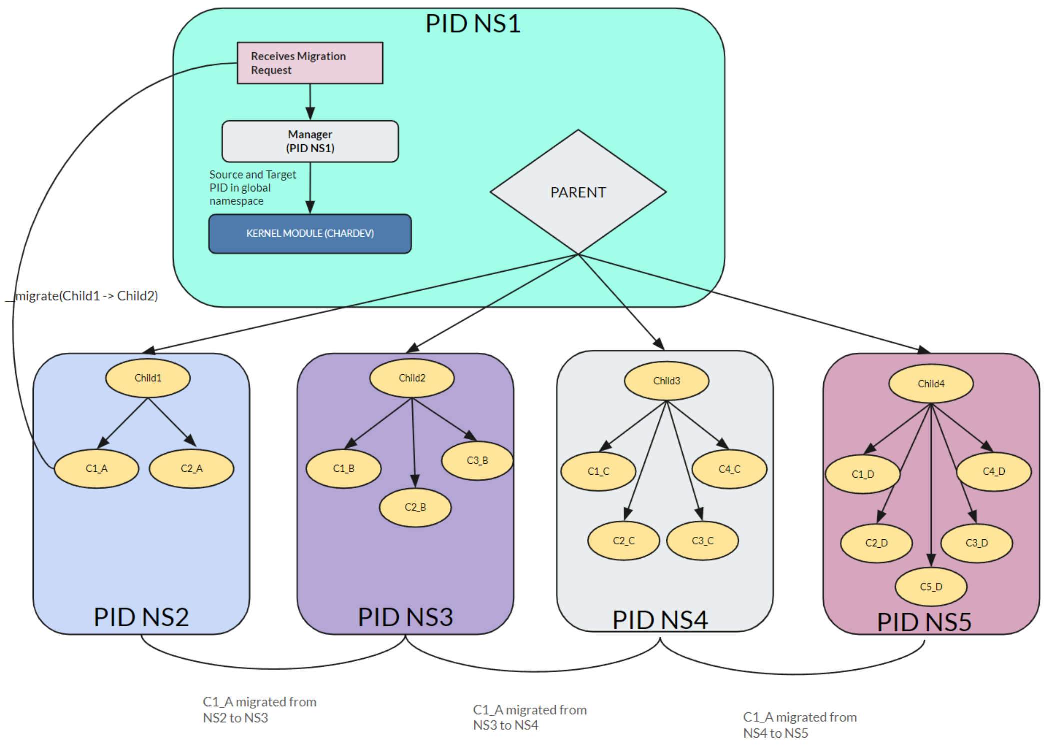
PID NS1
Receives Migration Request
Manager (PID NS1)
Source and Target PID in global namespace
KERNEL MODULE (CHARDEV)
PARENT
migrate(Child1 -> Child2)
Child1
C1_A
C2_A
PID NS2
Child2
C1_B
C2_B
C3_B
PID NS3
Child3
C1_C
C2_C
C3_C
C4_C
PID NS4
Child4
C1_D
C2_D
C3_D
C4_D
C5_D
PID NS5
C1_A migrated from NS2 to NS3
C1_A migrated from NS3 to NS4
C1_A migrated from NS4 to NS5

Figure 7: *Experimentation Setup*

VII Future Works

- The project can extended to improve upon the synchronization techniques. For example, currently, there is no way of knowing when the parent has actually completed the wait system call and the child is now detached. A signalling mechanism can be added to convey the kernel module of completion of wait and then we can move forward with attach phase.

- We have only handled the memory state of the migrating process, the remaining subsystems like the file system, etc. are not being migrated. Hence any open files would then become invalid after migration. These subsystems can be handled in the future by building upon the current design.

References

[1] *Linking Namespace at runtime*. URL: `https://patentimages.storage.googleapis.com/8b/28/85/25da70cdd5952e/US11681535.pdf`.

[2] *Security Namespace*. URL: `https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-sun.pdf`.

[3] *Change Root*. URL: `http://man7.org/linux/man-pages/man2/chroot.2.html/`.

[4] *Break out of chroot jail*. URL: `https://web.archive.org/web/20160127150916/http://www.bpfh.net/simes/computing/chroot-break.html/`.

[5] *Is chroot a security feature*. URL: `https://access.redhat.com/blogs/766093/posts/1975883/`.

[6] *Namespaces Types and API*. URL: `https://8gwifi.org/docs/linux-namespace.jsp`.

[7] *PID NS Hierarchy*. URL: `https://medium.com/geekculture/linux-namespaces-container-technology-a09da0813247`.