

سیستم‌های عامل

دکتر جلیلی

۴۰۱۱۰۶۳۲۸ - نیکا قادری
بهار ۱۴۰۴

فاز دوم

Container runtime system

تاریخ گزارش: ۱۰ تیر ۱۴۰۴

فهرست مطالب

۱	مقدمه
۲	فاز ۱
۳	۱.۲ محفظه چیست و اجزای آن ۱.۱.۲ اجزای یک محفظه ۲.۱.۲ Daemonless سیستم چیست؟ ۲.۲ namespace های لینوکس ۳.۲ گروههای کنترل (cgroups) ۴.۲ Chroot ۵.۲ سیستم‌های فایل (OverlayFS) Union ۶.۲ eBPF ۷.۲ ابزارها و محیط اجرایی ۸.۲ زیرساخت لینوکس ۹.۲ ابزارهای سیستم ۱۰.۲ محیط توسعه
۴	۱.۳ نحوه پیاده‌سازی ۱.۱.۳ پیاده‌سازی Namespace ها ۲.۱.۳ پیاده‌سازی سیستم فایل OverlayFS Union با ۳.۱.۳ پیاده‌سازی Jail Chroot ۴.۱.۳ پیاده‌سازی مدیریت منابع با cgroups v2 ۵.۱.۳ پیاده‌سازی قابلیت‌های پیشرفته ۶.۱.۳ پیاده‌سازی رابط خط فرمان (CLI) و چرخه حیات محفظه ۷.۱.۳ پیاده‌سازی نظارت با eBPF ۸.۱.۳ کد و توابع فایل‌های اصلی ۹.۱.۳ تعاریف و ثابت‌های سراسری ۱۰.۱.۳ تابع کمکی ۱۱.۱.۳ do_run ۱۲.۱.۳ do_list ۱۳.۱.۳ do_status ۱۴.۱.۳ تابع do_thaw و do_freeze ۱۵.۱.۳ تابع do_stop ۱۶.۱.۳ تابع do_start ۱۷.۱.۳ تابع do_rm ۱۸.۱.۳ تابع main ۱۹.۱.۳ اسکریپت setup_rootfs.sh ۲۰.۱.۳ اسکریپت monitor.py ۲۱.۱.۳ نتایج آزمایش‌ها
۵	۱.۳ آزمایش ۱: تأیید معماری بدون دیمون (Daemonless) ۲.۳ آزمایش ۲: ایزووله‌سازی فایل سیستم با chroot ۳.۳ آزمایش ۳: سیستم فایل OverlayFS Union با ۴.۳ آزمایش ۴: ایزووله‌سازی جامع Namespace ها ۵.۳ آزمایش ۵: مدیریت منابع با cgroups ۶.۳ آزمایش ۶: دستورات CLI و چرخه حیات محفظه ۷.۳ آزمایش ۷: توقف موقت با thaw و freeze ۸.۳ آزمایش ۸: ارتباط بین محفظه‌ها با اشتراک‌گذاری فضای نام IPC ۹.۳ آزمایش ۹: انتشار رویدادهای Mount با --propagate-mount

۵۳	۱۰.۳.۳ آزمایش ۱۰: قفل کردن روی هسته CPU و زمان‌بندی Round-Robin
۵۵	۱۱.۳.۳ آزمایش ۱۱: نظارت بر فرآخوانی‌های سیستمی با eBPF
۵۸	۴.۳ نقد و بررسی سیستم و پیشنهاداتی جهت بهبود
۵۹	۵.۳ نتیجه‌گیری
۶۰	۶.۳ منابع و مراجع

لطفاً توجه داشته باشید که تمامی کدهای برنامه و فایل‌های پروژه در آدرس گیت‌هاب زیر موجود هستند:
<https://github.com/NikaGhaderi/Container-Runtime-System>

چکیده

این گزارش، فرآیند طراحی، پیاده‌سازی و آزمایش یک سیستم مدیریت محفظه‌ی کامل با معماری بدون دیمون (*daemonless*) را تشریح می‌کند. این سیستم که با الهام از ابزارهای مدرنی مانند *Podman* و با استفاده از زبان برنامه‌نویسی *C* توسعه یافته، با تعامل مستقیم با هسته لینوکس، قابلیت ایجاد محیط‌های اجرایی ایزوله را فراهم می‌آورد. در این پروژه، فناوری‌های بنیادی کانتینری‌سازی شامل فضاهای نام *namespaces* برای ایزوله‌سازی منابع، گروه‌های کنترل *cgroups* برای مدیریت و محدودسازی منابع، سیستم فایل *OverlayFS* برای ایجاد ایمیج‌های لایه‌ای و کارآمد، و *chroot* برای ایجاد زندان فایل سیستم، به صورت عملی پیاده‌سازی شده‌اند. نتیجه‌ی نهایی، یک ابزار خط فرمان *CLI* کارا است که چرخه‌ی حیات کامل یک محفظه، از ایجاد و اجرا گرفته تا توقف، راماندازی مجدد و حذف را مدیریت می‌کند. همچنین، قابلیت‌های پیشرفت‌های مانند اشتراک‌گذاری منابع بین محفظه‌ها، بهینه‌سازی عملکرد با قفل کردن پردازه روی هسته *CPU*، و نظارت بر فراخوانی‌های سیستمی با استفاده از *eBPF* پیاده‌سازی و با موفقیت آزمایش شده‌اند.

کلمات کلیدی: مدیریت محفظه، کانتینر، لینوکس، معماری بدون دیمون، فضاهای نام، *eBPF*, *OverlayFS*, *cgroups*, *namespaces*، میانجی.

۱ مقدمه

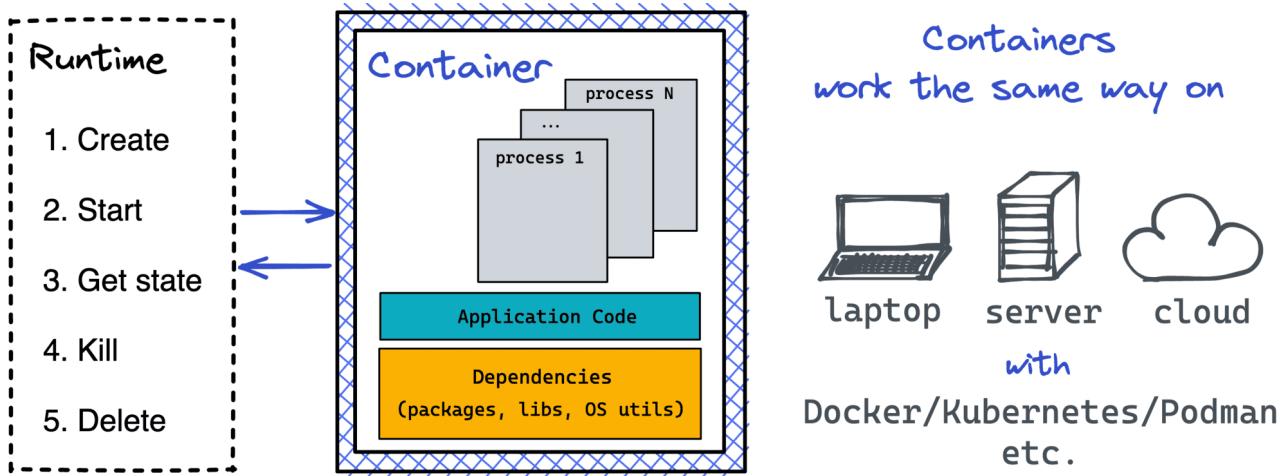
در دنیای مدرن رایانش ابری و توسعه‌ی نرم‌افزار، محفظه‌ها (*Containers*) به یکی از فناوری‌های کلیدی و تحول‌آفرین تبدیل شده‌اند. این فناوری با فراهم کردن امکان بسته‌بندی و اجرای برنامه‌ها در محیط‌های ایزوله، سیک و قابل حمل، مشکلات مربوط به وابستگی‌ها و تفاوت‌های محیطی را برطرف کرده و توسعه، استقرار و مقیاس‌پذیری نرم‌افزار را به شکل چشمگیری ساده‌سازی نموده است. سیستم عامل لینوکس، به عنوان بستر اصلی این فناوری، مجموعه‌ای از قابلیت‌های قادرمند در سطح هسته مانند فضاهای نام *cgroups*, *namespaces*, *OverlayFS*، و سیستم‌های فایل (*Union*) را ارائه می‌دهد که پایه‌های اصلی ساخت یک محفظه را تشکیل می‌دهد.

این گزارش، مستندسازی کامل مراحل طراحی، پیاده‌سازی و اعتبارسنجی یک سیستم مدیریت محفظه است که با الهام از معماری مدرن و بدون دیمون (*daemonless*) ابزارهایی مانند *Podman* توسعه یافته است. برخلاف معماری‌های سنتی که به یک پروسه‌ی پس‌زمینه‌ی دائمی برای مدیریت محفظه‌ها وابسته‌اند و یک نقطه‌ی شکست واحد (*single point of failure*) ایجاد می‌کنند، این پروژه با تعامل مستقیم با فراخوانی‌های سیستمی هسته لینوکس، یک راهکار پایدارتر و سبک‌تر ارائه می‌دهد.

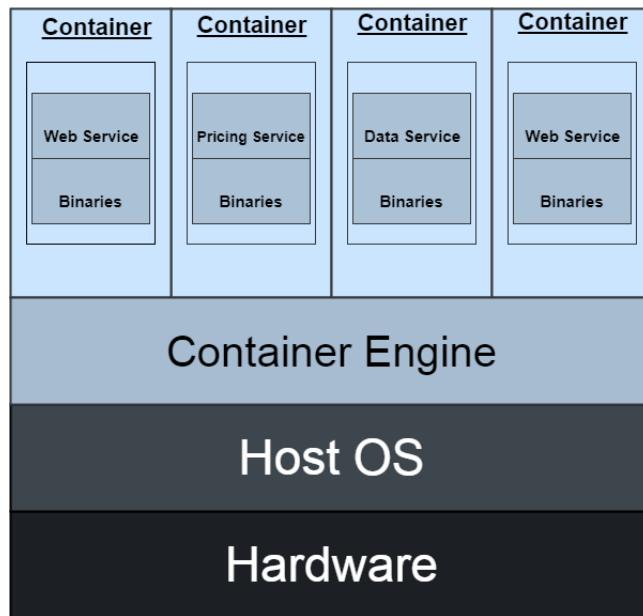
در فاز اول این پروژه، مبانی نظری و ابزارهای مورد نیاز شناسایی شدند. در فاز دوم که در این گزارش به تفصیل شرح داده می‌شود، این مفاهیم به صورت عملی پیاده‌سازی شده و یک ابزار خط فرمان *CLI* کامل برای مدیریت چرخه‌ی حیات محفظه‌ها توسعه یافته است. این گزارش ابتدا به تشریح دقیق نحوه‌ی پیاده‌سازی هر یک از قابلیت‌ها، از جمله ایجاد انواع *namespace*ها، *cgroups*، راماندازی *OverlayFS* و استفاده از *eBPF* برای نظارت می‌پردازد. سپس، با ارائه مجموعه‌ای از گزارش‌های تست جامع، صحت عملکرد هر یک از این قابلیت‌ها، از ایزوله‌سازی پایه گرفته تا مدیریت منابع و قابلیت‌های پیشرفت‌های پیش‌گفته، به صورت عملی به اثبات می‌رسد. در نهایت، با یک نقد و بررسی صادقانه، به محدودیت‌های سیستم فعلی و مسیرهای ممکن برای توسعه و بهبود آن در آینده پرداخته خواهد شد.

۱.۲ محفظه چیست و اجزای آن

محفظه (Container) یک بسته اجرایی سبک وزن و مستقل است که شامل تمامی موارد لازم برای اجرای یک نرم افزار، از جمله کد، ابزارهای سیستم، کتابخانه‌ها و تنظیمات است. محفوظه‌ها از یکدیگر و از سیستم میزبان جدا شده‌اند، که این امر استفاده بهینه از منابع و رفتار یکسان در محیط‌های مختلف را فراهم می‌کند. این فناوری بر پایه تکنولوژی‌هایی مانند *cgroup* و *namespace* در لینوکس بنا شده است و به کاربران این امکان را می‌دهد تا برنامه‌های خود را در محیط‌های ایزوله اجرا کنند.



شکل ۱: عملکرد کانتینرهای در یک نگاه



شکل ۲: این تصویر نشان می‌دهد که چگونه چندین برنامه کانتینری شده (App A to App F) بر روی یک *container engine* (مثل Docker) سیستم عامل میزبان و زیرساخت قرار می‌گیرند. *container engine* به عنوان یک لایه میانی، امکان اجرای این برنامه‌ها را به صورت مجزا و بسته‌بندی شده فراهم می‌کند.

۱.۱.۲ اجزای یک محفظه

۱. Runtime

ران تایم کانتینر نرم افزاری است که کانتینرهای را روی سیستم عامل اجرا و مدیریت می‌کند. وظایف اصلی آن شامل دریافت تصاویر، ایجاد محیط ایزوله با استفاده از قابلیت‌های هسته، اجرای فرآیند کانتینر و مدیریت چرخه حیات آن است. ران تایم، پلی بین پلتفرم‌های مدیریت کانتینر و هسته سیستم عامل برای اجرای صحیح و مجزای برنامه‌ها در کانتینرهای است. مثال‌هایی از *Podman*: *runtime container* و *Docker Engine* (که از *Contained Docker Engine* استفاده می‌کند) و *Containerized Docker Engine*.

۲: Container image

یک الگوی^۱ فقط خواندنی^۲ که شامل کد برنامه، runtime، کتابخانه‌ها، وابستگی‌های لازم برای اجرای برنامه درون یک محفظه است. تصاویر از مجموعه‌ای از دستورات در یک Dockerfile یا با استفاده از ابزارهای دیگر مانند Podman build ساخته می‌شوند و به صورت لایه‌بندی شده^۳ ذخیره می‌شوند. هر تصویر همچنین شامل یک مانیفست^۴ است که اطلاعاتی درباره لایه‌ها و پیکربندی تصویر را در بر دارد. تصاویر کانتینر معمولاً در رجیستری‌های کانتینر^۵ ذخیره و توزیع می‌شوند.

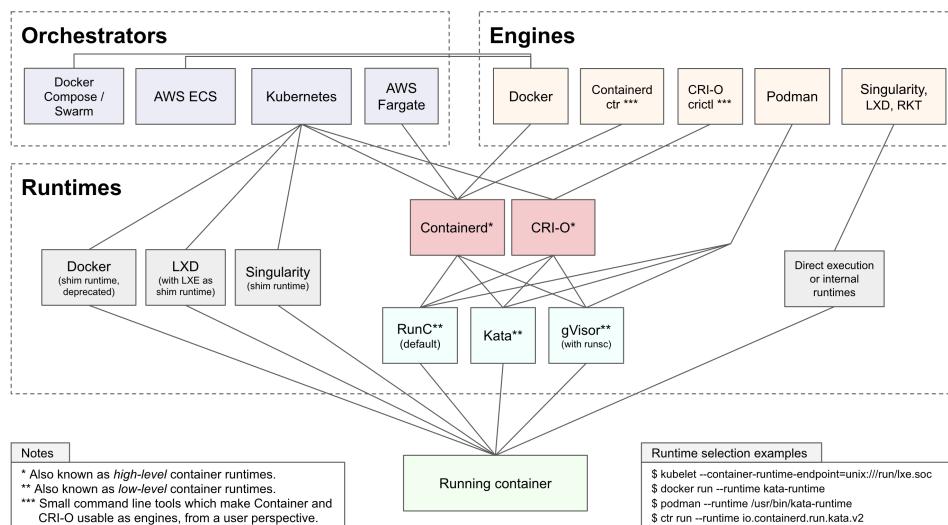
۳ Container Engine

ابزاری است که به عنوان یک رابط کاربری سطح بالا (High-Level Interface) عمل کرده و امکان تعامل با ران‌تايم‌های کانتینر را فراهم می‌کند. موتور محفظه مسئول مدیریت چرخه حیات کانتینرها (ایجاد، اجرا، توقف، حذف)، مدیریت تصاویر کانتینر (pull, push, build)، مدیریت شبکه‌ها و حجم‌های داده مربوط به کانتینرها است. برای مثال، CLI Docker runtime Podman است و هم موتور است.

در واقع موتور محفظه، ابزاری سطح بالا است که مسئول مدیریت چرخه حیات کانتینرها، تصاویر، شبکه‌ها و حجم‌های داده می‌باشد. این ابزار به عنوان یک رابط کاربری عمل کرده و دستورات کاربران را برای مدیریت زیرساخت کانتینری دریافت می‌کند. در مقابل، ران‌تايم کانتینر، جزء سطح پایین‌تری است که وظیفه اجرای واقعی کانتینرها بر روی سیستم عامل میزبان را بر عهده دارد. ران‌تايم با هسته سیستم عامل تعامل داشته و از قابلیت‌های آن برای ایجاد جداسازی و تخصیص منابع لازم برای اجرای کانتینرها استفاده می‌کند. به طور خلاصه، موتور محفظه مدیریت کلی را انجام می‌دهد و ران‌تايم، اجرای مستقیم کانتینرها را بر اساس دستورات موتور بر عهده دارد.

۴ Orchestrator

ابزاری که محفظه‌های متعدد را در سرورهای متعدد مدیریت می‌کند. Kubernetes محبوب‌ترین orchestrator است، اما orchestra-tor دیگری مانند Mesos Apache Swarm و Docker وجود دارد.



شکل ۳: نمایی از معماری کانتینر: ارتباط بین Orchestrator‌ها، Engines و Runtimes برای اجرای یک کانتینر.

۲.۱.۲ سیستم Daemonless چیست؟

سیستم daemonless، در زمینه مدیریت محفظه، به سیستمی گفته می‌شود که هیچ فرآیند پس‌زمینه (daemon) در آن برای مدیریت محفظه‌ها اجرا نمی‌شود. در سیستم‌های سنتی مانند Docker، یک daemon در پس‌زمینه اجرا می‌شود تا عملیات محفظه را مدیریت کند. اما در سیستم‌مانند Podman daemonless، عملیات محفظه‌ها مستقیماً توسط کاربر یا از طریق یک ابزار client انجام می‌شود، بدون نیاز به daemon. این معناری مفید است زیرا نقطه تکی شکست را حذف کرده و انعطاف‌پذیری بیشتری در مدیریت محفظه‌ها فراهم می‌کند.

۲.۲ namespace های لینوکس

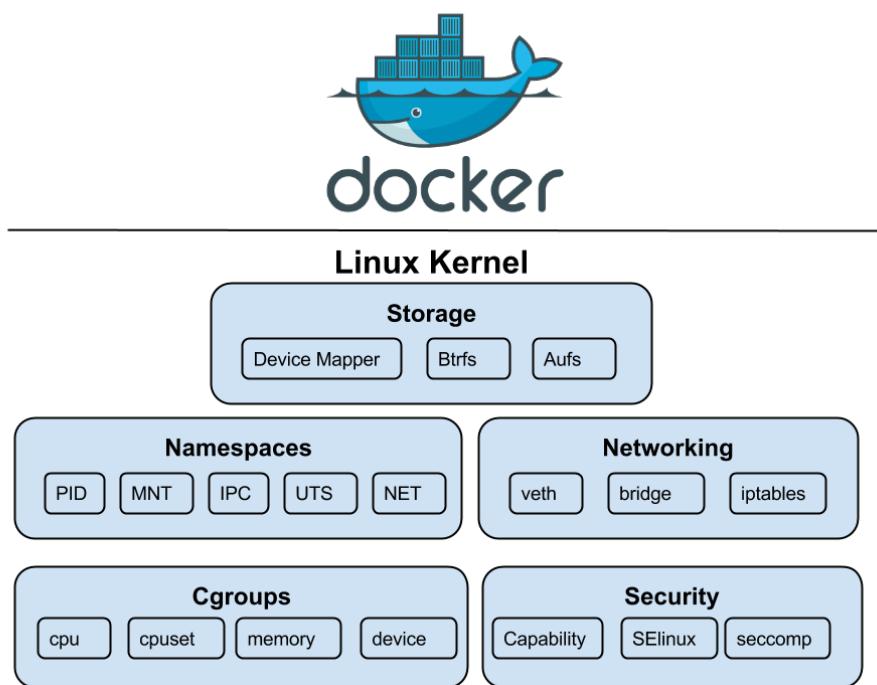
نامهای لینوکس ویرگی‌ای از هسته لینوکس هستند که منابع سیستمی را به گونه‌ای تقسیم‌بندی می‌کنند که هر مجموعه از فرآیندها تنها به منابع اختصاصی خود دسترسی داشته باشند. این ایزو‌لاسیون برای محفظه‌سازی حیاتی است، زیرا امکان اجرای چندین محفظه روی یک میزبان را بدون تداخل فراهم می‌کند. لازم به ذکر است که ایزو‌لاسیون کامل یک محفظه از همکاری چندین نوع namespace حاصل می‌شود. همچنین، namespacesها به همراه cgroups برای دستیابی به ایزو‌لاسیون و مدیریت منابع به صورت جامع عمل می‌کنند.

نوع namespacel از عبارتند از:

- **مانت (Mount):** نقاط مانت سیستم فایل را ایزوله می‌کند، به طوری که هر محفظه دید مستقل و جدایی از ساختار فایل‌ها دارد. این امکان را می‌دهد که تغییرات در سیستم فایل یک محفظه بر سایر محفظه‌ها یا میزبان تأثیر نگذارد.

template^۱
read – only^۲
layered^۳
manifest^۴
container registries^۵

- **UTS**: نام میزبان و دامنه را ایزوله می‌کند، درنتیجه محفظه می‌تواند هویت مستقلی داشته باشد.
- **IPC**: منابع ارتباط بین فرآیندی مانند حافظه اشتراکی و صفحه‌ای پیام را ایزوله می‌کند، که از ارتباط مستقیم فرآیندهای محفظه‌های مختلف جلوگیری می‌کند.
- **شبکه (Network)**: رابطها و پشته شبکه (آدرس‌های IP و جداول مسیریابی) را ایزوله می‌کند، به هر محفظه امکان داشتن تنظیمات شبکه مستقل می‌دهد.
- **PID**: شناسه‌های فرآیند را ایزوله می‌کند، به طوری که هر محفظه فضای PID مستقل با فرآیند ابتدایی (1) دارد، انگار یک سیستم عامل مستقل داریم. همچنین، شایان ذکر است که *PID namespace* ها می‌توانند سلسله مراتبی باشند.
- **کاربر (User)**: شناسه‌های کاربری و گروهی را ایزوله می‌کند و امکان جداسازی امتیازات را فراهم می‌کند. این *namespace* در زمینه امنیت بسیار مهم است و به فرآیندهای غیرمجاز داخل کانتینر اجازه می‌دهد تا به عنوان root در داخل عمل کنند (بدون داشتن امتیاز root در میزبان).
- **زمان (Time)**: زمان سیستم را ایزوله می‌کند، به محفظه‌ها امکان داشتن ساعت‌های مستقل می‌دهد. برای مثال، یک برنامه تست ممکن است به یک ساعت سیستم خاص نیاز داشته باشد که با زمان واقعی سیستم تداخل داشته باشد. این *namespace*ها با همکاری یکدیگر، ایزولاسیون لازم برای اجرای مستقل محفظه‌ها را فراهم می‌کنند.



شکل ۴: معماری کانتینر در لینوکس: نقش کلیدی (PID, MNT, IPC, UTS, NET) (namespace) ها در فراهم کردن ایزولاسیون برای Kubernetes و Docker بر بستر هسته لینوکس.

۳.۲ گروههای کنترل (cgroups)

گروههای کنترل (cgroups) ویژگی‌ای از هسته لینوکس هستند که امکان سازماندهی فرآیندها به صورت سلسله‌مراتبی و کنترل و محدودسازی میزبان استفاده آن‌ها از منابع سیستمی مانند CPU، حافظه، ورودی/خروجی (I/O)، پهنای باند شبکه وغیره را فراهم می‌کنند. *cgroup*ها ابزاری بنیادین برای مدیریت منابع در محیط‌های کانتینری و همچنین سایر موارد مانند مجازی‌سازی و مدیریت کیفیت سرویس (QoS) در سیستم‌های لینوکسی به شمار می‌روند.

به طور خلاصه، *cgroup*ها کاربردهای زیر را دارند:

- محدود کردن منابع: تعیین حداقل میزان استفاده از هر منبع (مانند حداکثر حافظه قابل استفاده توسط یک گروه از فرآیندها).
- اولویت‌بندی: تخصیص سهم نسبی از منابع به گروههای مختلف (مانند تخصیص سهم بیشتری از CPU به یک کانتینر مهم).
- اندازه‌گیری و گزارش‌گیری: پیگیری میزان مصرف منابع توسط هر گروه.
- کنترل و انجام: متوقف کردن و از سرگیری فرآیندهای یک گروه.

در زمینه کانتینرها، *cgroup*ها به دلایل زیر در سیستم نقشی حیاتی ایفا می‌کنند:

- تضمین تخصیص منصفانه منابع: اطمینان حاصل می‌کنند که هر کانتینر تنها از منابع تخصیص یافته خود استفاده می‌کند و یک کانتینر پر مصرف نمی‌تواند منابع کل سیستم را اشغال کند.
- ایجاد محیط‌های قابل پیش‌بینی: با محدود کردن منابع، عملکرد کانتینرها قابل پیش‌بینی تر می‌شود و از تاثیر نوسانات بار در سایر کانتینرها جلوگیری می‌شود.
- بهبود چگالی (Density): امکان اجرای تعداد بیشتری کانتینر بر روی یک میزبان واحد را با مدیریت کارآمد منابع فراهم می‌کنند.

دو نسخه اصلی از *cgroup* وجود دارد:

- *cgroups v1*: در لینوکس ۲.۶.۲۴ معرفی شد و از یک معماری چندسلسله‌مراتبی (*multi-hierarchy*) استفاده می‌کند. در این نسخه، کنترلرهای مختلف (مانند کنترل حافظه، *CPU*، *I/O*) می‌توانند در سلسله‌مراتب‌های جداگانه نصب شوند. این انعطاف‌پذیری گاهی منجر به پیچیدگی در مدیریت و ناسازگاری بین کنترلرها می‌شده.
- *cgroups v2*: در لینوکس ۳.۱۰ معرفی شد و در نسخه ۴.۵ رسمی شد، با هدف ساده‌سازی مدیریت و رفع مشکلات نسخه اول، یک معماری سلسله‌مراتبی واحد (*single unified hierarchy*) را ارائه می‌دهد. *cgroups v2* مدل منسجم‌تری برای مدیریت منابع فراهم کرده و عملکرد و کارایی بهتری را ارائه می‌دهد.

Chroot ۴.۲

یک فراخوانی سیستمی (*syscall*) است که دایرکتوری ریشه (/) یک فرآیند در حال اجرا و تمام فرزندان آن را به یک دایرکتوری مشخص شده جدید تغییر می‌دهد. پس از انجام عمل *chroot*، فرآیند و فرزندان آن تصور می‌کنند که دایرکتوری جدید، ریشه سیستم فایل آن‌ها است و نمی‌توانند به فایل‌ها و دایرکتوری‌های خارج از این "ریشه جدید" دسترسی پیدا کنند. هدف اصلی *chroot* ایجاد یک نوع ایزو‌لایشن سیستم فایل است. این مکانیسم به منظور محدود کردن دسترسی یک برنامه یا کاربر به بخش خاصی از سیستم فایل به کار می‌رود و می‌تواند برای اهداف مختلفی مانند موارد زیر مفید باشد:

- محیط‌های تست و توسعه: ایجاد یک محیط ایزو‌لایه برای تست نرم‌افزار بدون خطر آسیب رساندن به سیستم اصلی.
 - امنیت: محدود کردن دسترسی برنامه‌های بالقوه خطرناک به بخش‌های حساس سیستم فایل.
 - ایجاد محیط‌های شبیه‌سازی شده: فراهم کردن یک سیستم فایل کوچک و مستقل برای اجرای برنامه‌های خاص.
- با وجود مزایای ایزو‌لایشن سیستم فایل، *chroot* دارای محدودیت‌های قابل توجهی در زمینه کانتینر سازی مدرن است:
- عدم ایزو‌لایشن سایر منابع: *chroot* سایر منابع سیستمی مانند فرآیندها (*PID*)، شبکه، حافظه، دستگاه‌ها (مانند */dev*) و ارتباطات بین فرآیندی (*IPC*) را ایزو‌لایه نمی‌کند. یک فرآیند *chroot* همچنان می‌تواند با سایر فرآیندهای سیستم تعامل داشته باشد، به شبکه دسترسی پیدا کند و دستگاه‌ها را مدیریت کند.
 - عدم وجود لایه‌بندی (*Stacking*): به صورت یک لایه عمل می‌کند و امکان ایجاد سیستم‌های فایل لایه‌ای (مانند آنچه در تصاویر کانتینر استفاده می‌شود) را فراهم نمی‌کند.
 - فرار از *chroot*: در برخی موارد، با استفاده از فراخوانی‌های سیستمی خاص یا سوءاستفاده از پیکربندی‌های نادرست، امکان فرار از محیط *chroot* و دسترسی به سیستم فایل اصلی وجود دارد، به ویژه اگر فرآیند داخل *chroot* دارای امتیازات *root* باشد.

در زمینه محفظه‌ها (*Containers*، *chroot* به تنهایی برای ایجاد ایزو‌لایشن جامع کافی نیست. به همین دلیل، در سیستم‌های کانتینری مدرن، معمولاً در ترکیب با *namespace*‌های لینوکس و *cgroup*‌ها استفاده می‌شود. *Namespace*‌ها ایزو‌لایشن منابع مختلف سیستم (مانند *PID*، شبکه، *IPC*) را فراهم می‌کنند، در حالی که داده‌های زیرین به صورت فیزیکی جداگانه باقی مانند. در زمینه کانتینرها، سیستم‌های فایل *Union* نقش بسیار مهمی در مدیریت و اشتراک‌گذاری تصاویر کانتینر ایفا می‌کنند. *Cgroup*‌ها نیز برای مدیریت و محدودسازی مصرف منابع توسط کانتینرها به کار می‌روند.

5.۲ سیستم‌های فایل (*OverlayFS Union*)

سیستم‌های فایل *Union* دسته‌ای از سیستم‌های فایل هستند که قابلیت ادغام چندین شاخه (دایرکتوری) را به صورت یک سیستم فایل مجازی واحد فراهم می‌کنند. این تکنیک امکان ارائه یک دید یکپارچه از داده‌ها را فراهم می‌سازد، در حالی که داده‌های زیرین به صورت فیزیکی جداگانه باقی مانند. در زمینه کانتینرها، سیستم‌های فایل *Union* نقش بسیار مهمی در مدیریت و اشتراک‌گذاری تصاویر کانتینر ایفا می‌کنند. *OverlayFS* یکی از محبوب‌ترین و کارآمدترین سیستم‌های فایل *Union* در لینوکس می‌باشد. در پیاده‌سازی های کانتینر مانند *Docker* و *Podman* مورد استفاده قرار می‌گیرد. از منظر ساختار منطقی، *OverlayFS* بر پایه تعامل چند دایرکتوری اساسی استوار است:

- لایه پایین‌تر (*Lower Layer*): این دایرکتوری یا دایرکتوری‌ها معمولاً به صورت فقط‌خواندنی هستند و محتواهای تصویر پایه کانتینر و همچنین لایه‌های میانی تصاویر را شامل می‌شوند. این لایه‌ها می‌توانند به صورت مشترک بین چندین کانتینر کانتینر و تصویر استفاده شوند، که منجر به صرفه‌جویی قابل توجهی در فضای دیسک و زمان انتقال تصاویر می‌شود.
- لایه بالاتر (*UpperLayer*): این دایرکتوری به صورت قابل نوشتن^۶ است و به هر کانتینر یک فضای مجزا برای اعمال تغییرات در سیستم فایل ارائه می‌دهد. تمام تغییرات ایجاد شده در طول عمر یک کانتینر (مانند ایجاد، حذف یا ویرایش فایل‌ها) در این لایه ذخیره می‌شوند.

⁶writable

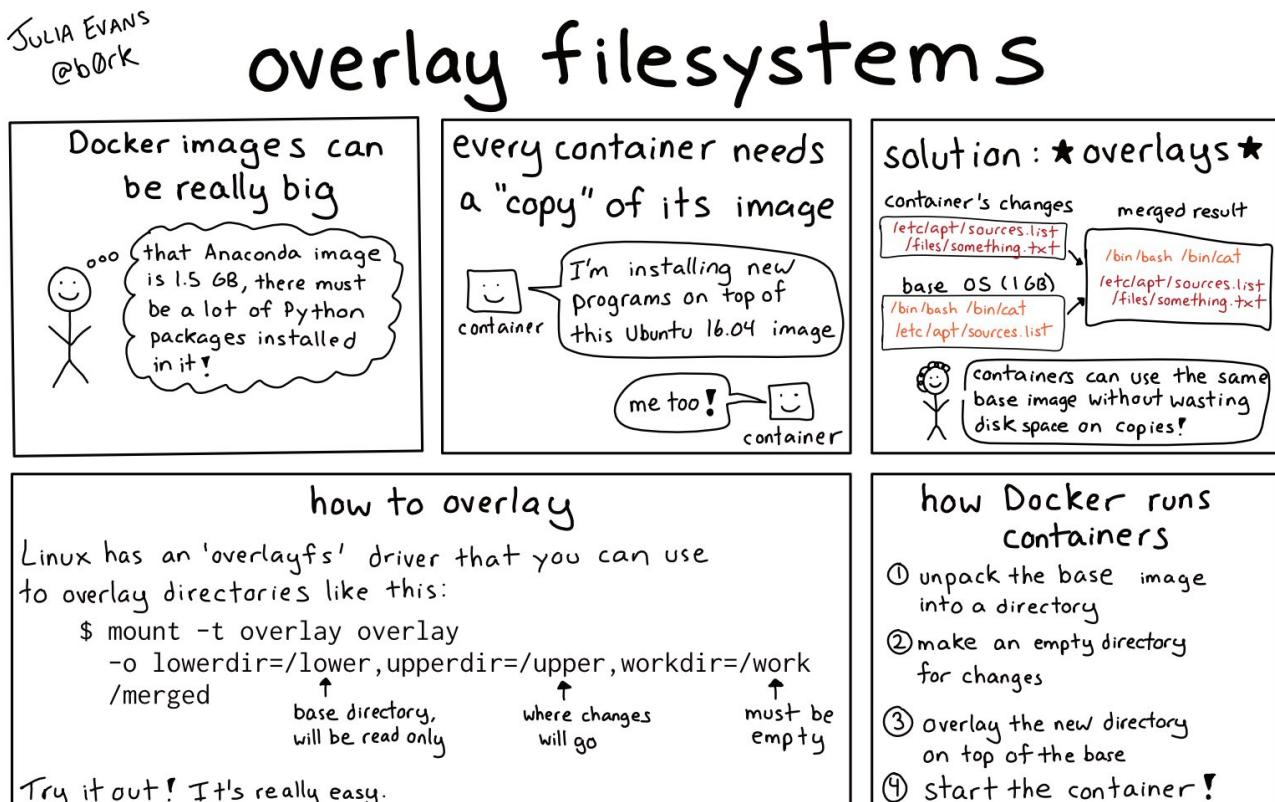
- لایه کاری (Work Directory): یک دایرکتوری داخلی است که توسط *OverlayFS* برای ردیابی تغییرات و انجام عملیات – *copy* استفاده می‌شود.

- دایرکتوری ادغام شده (Merged Directory): این یک نمای مجازی است که محتوای لایه‌های پایین‌تر و بالاتر را به صورت یک سیستم فایل واحد و یکپارچه به کانتینر ارائه می‌دهد. هنگامی که یک کانتینر به فایلی دسترسی پیدا می‌کند، *OverlayFS* ابتدا در لایه بالاتر و سپس در لایه‌های پایین‌تر به دنبال آن می‌گردد.

اهمیت *OverlayFS* (و سایر سیستم‌های فایل Union) در کانتینرها:

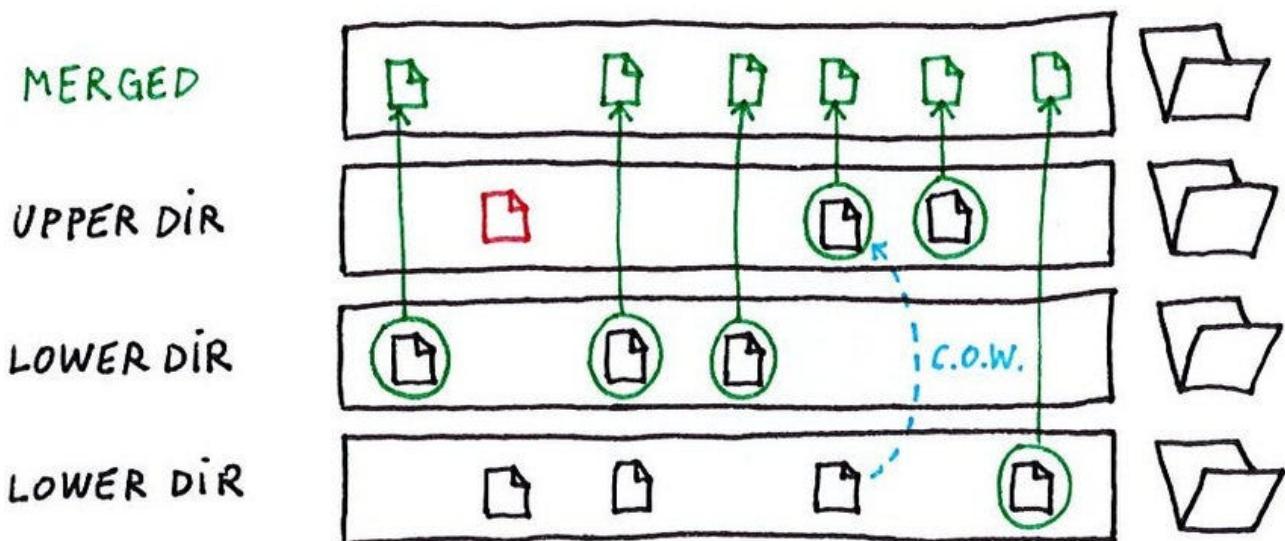
- اشتراک‌گذاری لایه‌ها: امکان اشتراک‌گذاری لایه‌های فقط خواندنی بین چندین کانتینر و تصویر را فراهم می‌کند، که منجر به کاهش مصرف دیسک و تسریع فرآیند استقرار کانتینرها می‌شود.
- لایه‌های قابل‌نوشتن مجزا برای هر کانتینر: هر کانتینر یک لایه قابل‌نوشتن مجزا دارد که تغییرات آن را از سایر کانتینرها و تصویر پایه جدا نگه می‌دارد. این امر اطمینان می‌دهد که تغییرات یک کانتینر بر سایرین تأثیر نمی‌گذارد.
- مکانیسم *Copy-on-Write*: هنگامی که یک کانتینر قصد نوشتن در یک فایل موجود در لایه‌های پایین‌تر را دارد، این کار باعث حفظ یکپارچگی لایه‌های پایین‌تر (تصویر پایه) می‌شود.
- ساخت سریع تصاویر: فرآیند ساخت تصاویر کانتینر با استفاده از لایه‌ها تسریع می‌یابد، زیرا لایه‌های تغییرنیافته می‌توانند از تصاویر قبلی به ارت برستند.

علاوه بر *OverlayFS*، سیستم‌های فایل *Union* دیگری مانند *AUFS* و *Btrfs* نیز در گذشته یا در موارد خاص برای کانتینرها استفاده شده‌اند، اما *OverlayFS* به دلیل عملکرد و سادگی، به راهکار غالب تبدیل شده است. به طور خلاصه، سیستم‌های فایل *Union* و به طور خاص *OverlayFS*، یک جزء حیاتی در معماری کانتینرها هستند که امکان مدیریت کارآمد تصاویر، اشتراک‌گذاری لایه‌ها و فراهم کردن فضای قابل‌نوشتن مجزا برای هر کانتینر را فراهم می‌کنند.



شکل ۵: مشکل حجم بالای تصاویر Docker و راه حل *Overlay Filesystems* برای اشتراک‌گذاری لایه‌ها و کاهش مصرف دیسک.

eBPF (extended Berkeley Packet Filter) فناوری‌ای است که امکان اجرای برنامه‌های این در هسته لینوکس را بدون تغییر کد منع یا بارگذاری مازول فراهم می‌کند. برخلاف فیلتر سنتی برکلی (BPF) که عمدها برای فیلتر کردن ترافیک شبکه استفاده می‌شد، eBPF به طور قابل توجهی گسترش یافته و اکنون می‌تواند به نقاط مختلف هسته متصل شده و رویدادهای گوناگونی را ردیابی و دستکاری کند. برخی از کاربردهای کلیدی eBPF عبارتند از:



شکل ۶: نمایش بصری عملکرد OverlayFS: ادغام لایه‌های پایین‌تر (فقط خواندنی) و لایه بالاتر (قابل نوشتن) برای ایجاد یک دید یکپارچه

- ردگیری فراخوانی‌های سیستم (System Call Tracing): امکان نظارت دقیق بر فراخوانی‌های سیستم که توسط فرآیندها انجام می‌شوند، از جمله پارامترها، مقادیر بازگشته و مدت زمان اجرا. این قابلیت برای درک رفتار برنامه‌ها و شناسایی مشکلات عملکردی یا امنیتی بسیار ارزشمند است.
- مانیتورینگ رویدادهای هسته (Kernel Event Monitoring): قابلیت ردیابی رویدادهای مختلف در سطح هسته، مانند زمان‌بندی فرآیندها، مدیریت حافظه، فعالیت‌های ورودی/خروجی و غیره. این امکان دید عمیقی به عملکرد داخلی هسته فراهم می‌کند.
- فیلتر کردن پسته‌های شبکه (Network Packet Filtering): همچنان به عنوان یک فیلتر بسته بسیار کارآمد و انعطاف‌پذیر برای تجزیه و تحلیل و دستکاری ترافیک شبکه در سطح هسته عمل می‌کند.
- تجزیه و تحلیل عملکرد (Performance Analysis): جمع‌آوری دقیق عملکرد سیستم و برنامه‌ها با سربار کم، امکان شناسایی گلوگاه‌ها و بهینه‌سازی عملکرد را فراهم می‌کند.
- امنیت (Security): پیاده‌سازی سیاست‌های امنیتی سفارشی در سطح هسته، مانند تشخیص و جلوگیری از رفتارهای غیرعادی یا حملات امنیتی.

در این پروژه خاص، eBPF به عنوان ابزاری برای نظارت بر فراخوانی‌های سیستم مرتبط با ایجاد یا حذف cgroup و namespace کار گرفته می‌شود. با اتصال برنامه‌های eBPF به نقاط مناسب در هسته، می‌توان به طور بی‌درنگ از ایجاد یا حذف این سازوکارهای ایزو‌لاسیون مطلع شد و اطلاعات مربوطه را جمع‌آوری کرد. این رویدادها سپس در یک فایل ثبت می‌شوند تا امکان تجزیه و تحلیل و بررسی فعالیت‌های مربوط به مدیریت کانتینرها فراهم گردد. استفاده از eBPF در این زمینه امکان نظارت دقیق و با سربار کم را بدون ایجاد تغییر در کد برنامه‌ها یا هسته فراهم می‌کند، که یک مزیت قابل توجه در محیط‌های عملیاتی است.

۷.۲ ابزارها و محیط اجرایی

۱.۷.۲ زیرساخت لینوکس

- هسته لینوکس نسخه ۴/۱۵ یا بالاتر: نسخه‌های جدیدتر هسته لینوکس شامل پشتیبانی کامل از ویژگی‌های پیشرفته مانند user namespaces، بهبودهای cgroup v2، و قابلیت‌های eBPF می‌شوند. حداقل نسخه ۴/۱۵ به دلیل پشتیبانی پایدار از ویژگی‌های ضروری مانند:

- برای ایجاد محفظه‌ها بدون نیاز به دسترسی ریشه unprivileged user namespaces
- با معماری یکپارچه‌تر برای مدیریت منابع cgroup v2
- eBPF JIT compiler برای اجرای کارآمد برنامه‌های eBPF

- کتابخانه‌های ضروری:

- libcgroup: کتابخانه مدیریت cgroup‌ها که امکان ایجاد، پیکربندی و نظارت بر گروه‌های کنترل منابع را فراهم می‌کند.
- libcap: کتابخانه مدیریت قابلیت‌های فرایند (Capabilities) که امکان تفکیک دسترسی‌های ریشه را به قابلیت‌های ریزدانه‌تر فراهم می‌کند.

نکات پیکربندی: فعال‌سازی ویژگی‌های هسته از طریق فایل `:/boot/config - $(uname - r)`

```
CONFIG_CGROUPS = y •
CONFIG_OVERLAY_FS = y •
CONFIG_BPF_SYSCALL = y •
```

۴.۷.۲ ابزارهای سیستم

- `unshare`: ابزار خط فرمان برای ایجاد *namespace* جدید. پارامترهای کلیدی:
 - : جداسازی فضای *mount* —
 - : جداسازی درخت *PID* —

`nsenter -t PID -n ipaddrshow`: ورود به *namespace* موجود یک فرایند. مثال دسترسی به *namespace* شبکه:

- `cgcreate`: ایجاد سلسله مراتب *cgroup* جدید. مثال ایجاد محدودیت حافظه:
 - : ایجاد سیستم فایل اتحادی با معماری *OverlayFS*. ساختار لایه‌ها:
 - : لایه‌های فقط-خواندنی *lowerdir* —
 - : لایه نوشتی *upperdir* —

۴.۷.۳ محیط توسعه

- توزیع پیشنهادی: *Ubuntu ۲۲/۰۴ LTS* به دلایل:

— پشتیبانی بلندمدت تا سال ۲۰۳۲

— وجود بسته‌های از پیش کامپایل شده برای *eBPF*

- کامپایلر: نسخه ۱۱ یا بالاتر *GCC* به دلیل پشتیبانی از استاندارد *C17*

- ابزارهای اشکال‌زدایی:

— *strace*: ردگیری فراخوانی‌های سیستمی

— *bpftrace*: نوشتن اسکریپت‌های *eBPF*

- کتابخانه *eBPF*: نسخه ۱/۰ یا بالاتر *libbpf* شامل:

— *API*: پایدار برای بارگذاری برنامه‌ها

— *CO-RE*: پشتیبانی از

۱.۳ نحوه پیاده‌سازی

در این بخش، به تشریح دقیق رویکردها و تکنیک‌های استفاده شده برای پیاده‌سازی هر یک از قابلیت‌های اصلی سیستم مدیریت محفظه پرداخته می‌شود. تمرکز اصلی بر روی تعامل مستقیم با هسته لینوکس از طریق فراخوانی‌های سیستمی و رابطه‌های فایل‌سیستم مجازی بوده است.

۱.۳.۱ پیاده‌سازی Namespace ها

ایزوله‌سازی، هسته‌ی اصلی یک محفظه است و این امر از طریق فراخوانی سیستمی (`clone()` با پرچم‌های مناسب محقق شده است. در تابع `do_run` هنگام ایجاد یک پروسه‌ی فرزند جدید، مجموعه‌ای از پرچم‌ها به این فراخوانی ارسال می‌شود تا هر محفظه در `namespace`‌های مجازی خود ایجاد گردد.

:PID, UTS, Mount, Network Namespace •

برای ایزوله‌سازی این منابع، به ترتیب از پرچم‌های `CLONE_NEWWNET`, `CLONE_NEWNS`, `CLONE_NEWPID`, `CLONE_NEWUTS` استفاده شده است. پس از ایجاد `Network Namespace` جدید، این فضای فیزیکی را به طور پیش‌فرض هیچ رابط شبکه‌ای فعالی ندارد. بنابراین، در داخل تابع `container_main`، دستور `ip link set lo up` اجرا می‌شود تا رابط شبکه محلی `loopback` فعال شده و یک محیط شبکه ایزوله و کارا فراهم گردد.

:User Namespace •

این بخش با پرچم `CLONE_NEWUSER` فعال می‌شود. پیاده‌سازی این بخش با یک چالش کلیدی همراه است: (*race condition*). ممکن است پروسه‌ی فرزند قبل از آنکه نگاشت کاربران توسط والد تکمیل شود، شروع به کار کند. برای حل این مشکل، از یک (`pipe`) برای همگام‌سازی استفاده شده است. پروسه‌ی والد پس از اجرای (`clone()`، فایل‌های `uid_map` و `gid_map`) را در مسیر `/proc/[pid]` ایجاد می‌نماید تا کاربر `root` مجازی داخل محفظه (با `UID` صفر) را به کاربر عادی اجرا کننده دستور روی هاست نگاشت دهد. برای تشخیص صحیح کاربر اصلی هنگام استفاده از `sudo`, شناسه‌ها از متغیرهای محیطی `SUDO_UID` و `SUDO_GID` و `SUDO_USER` خوانده می‌شوند. پس از تکمیل موفقیت‌آمیز نگاشت، والد (`pipe`) را می‌بندد. پروسه‌ی فرزند در ابتدای اجرای خود منتظر بسته شدن این (`pipe`) می‌ماند و تنها پس از آن به ادامه‌ی کار خود می‌پردازد. این مکانیزم تضمین می‌کند که پروسه‌ی فرزند با مجوزهای صحیح کار خود را آغاز می‌کند.

:IPC Namespace •

به طور پیش‌فرض، پرچم `CLONE_NEWIPC` برای ایجاد یک فضای نام `IPC` خصوصی برای هر محفظه استفاده می‌شود. این کار از تداخل در منابعی مانند حافظه اشتراکی جلوگیری می‌کند.

۲.۱.۳ پیاده‌سازی سیستم فایل Union با OverlayFS

برای ایجاد یک سیستم فایل ایزوله و کارآمد برای هر محفظه، از `OverlayFS` استفاده شده است. پیاده‌سازی این بخش شامل مراحل زیر است:

۱. یک دایرکتوری پایه و فقط خواندنی به عنوان `lowerdir` در نظر گرفته می‌شود که همان ایمیج پایه ما (`ubuntu-base-image`) است.
۲. برای هر محفظه‌ی جدید، یک دایرکتوری منحصر به فرد به عنوان لایه قابل نوشتن `upperdir` و یک دایرکتوری کاری `workdir` در مسیر `overlay_layers/` ایجاد می‌شود.
۳. با استفاده از فراخوانی سیستمی (`mount`) و با تعیین نوع فایل‌سیستم به صورت `overlay`, این سه لایه با یکدیگر ادغام شده و در یک دایرکتوری `merged` مانند می‌شوند. این دایرکتوری ادغام شده، همان چیزی است که به عنوان ریشه‌ی فایل‌سیستم در اختیار محفظه قرار می‌گیرد.

۳.۱.۳ پیاده‌سازی Jail Chroot

پس از آنکه سیستم فایل `OverlayFS` با موفقیت در دایرکتوری `merged` مانند شد، از فراخوانی سیستمی (`chroot()`) استفاده می‌شود پروسه‌ی محفظه به فایل‌سیستم محدود شود. این فراخوانی، دایرکتوری ریشه‌ی پروسه را به مسیر `merged` تغییر می‌دهد. بلافاصله پس از آن، فراخوانی (`"/" chdir`) اجرا می‌شود تا اطمینان حاصل شود که دایرکتوری کاری فعلی نیز به ریشه‌ی جدید منتقل شده است. این دو عمل در کنار هم، یک "زندان" فایل‌سیستم ایجاد می‌کنند که پروسه قادر به مشاهده یا دسترسی به فایل‌های خارج از آن نیست.

۴.۱.۳ پیاده‌سازی مدیریت منابع با cgroups v2

برای کنترل و محدودسازی منابع، از رابط فایل‌سیستم `cgroups v2` استفاده شده است.

۱. ابتدا، یک سلسله‌مراقب اصلی برای برنامه‌ی ما در مسیر `/sys/fs/cgroup/my_runtime` / ایجاد می‌شود.
۲. برای هر محفوظه‌ی جدید، یک زیرگروه (`subgroup`) با نامی منحصر به فرد (مثلًا `[pid]_container`) در این مسیر ساخته می‌شود.
۳. سپس، محدودیت‌های درخواستی کاربر (که از طریق فلگ‌هایی مانند `--cpu` یا `--mem` دیافت شده) با نوشتن مقادیر مناسب در فایل‌های رابط آن `cgroup` اعمال می‌شوند. برای مثال، برای محدودیت حافظه، مقدار مورد نظر در فایل `memory.max` نوشته می‌شود.
۴. در نهایت، پروسه‌ی اصلی محفوظه در فایل `procs` آن زیرگروه نوشته می‌شود تا قوانین آن `cgroup` بر روی آن پروسه و تمام فرزندانش اعمال گردد.

این رویکرد، روش استاندارد و مدرن برای تعامل با `cgroups v2` است و نیازی به کتابخانه‌های جانبی ندارد.

۵.۱.۳ پیادهسازی قابلیت‌های پیشرفته

- اشتراک‌گذاری فضای نام IPC: این قابلیت با فلگ share-ipc--کنترل می‌شود. اگر این فلگ استفاده نشود، پرچم CLONE_NEWIPC به فراخوانی clone() اضافه شده و یک فضای نام خصوصی ایجاد می‌شود. اما در صورت استفاده از فلگ، این پرچم حذف شده و پروسه‌ی فرزند، فضای نام IPC والد خود (یعنی هاست) را به ارث می‌برد. این امر امکان ارتباط بین محفظه‌ها از طریق حافظه اشتراکی را فراهم می‌کند.
- انتشار رویدادهای Mount: قابلیت mount propagation با فلگ propagate-mount --فعال می‌شود. پیادهسازی آن دو مرحله دارد: ابتدا، دایرکتوری مشخص شده روی هاست با فراخوانی mount(NULL, path, NULL, MS_REC | MS_SHARED, NULL) به یک نقطه مانع اشتراکی تبدیل می‌شود. سپس، درون محفظه، یک دایرکتوری منتظر ایجاد شده و با یک (bind mount)، به نقطه مانع اشتراکی هاست متصل می‌شود. از این پس، هر رویداد مانع جدیدی که زیر آن دایرکتوری روی هاست رخ دهد، به صورت خودکار در محفظه نیز ظاهر خواهد شد.
- قفل کردن پردازه روی هسته (CPU Pinning): این قابلیت با فلگ pin-cpu--پیادهسازی شده و از دو فراخوانی سیستمی کلیدی استفاده می‌کند:
 1. sched_setaffinity(): این فراخوانی، پروسه‌ی محفظه را به یک هسته‌ی CPU خاص محدود می‌کند. هسته‌ی مورد نظر به صورت round-robin از میان هسته‌های موجود انتخاب می‌شود. شماره‌ی آخرین هسته‌ی استفاده شده در یک فایل موقت /tmp/my_runtime_next_cpu ذخیره می‌شود تا تخصیص‌ها بین محفظه‌های مختلف توزیع شود.
 2. sched_setscheduler(): این فراخوانی، سیاست زمان‌بندی پروسه را به SCHED_RR تغییر می‌دهد که یک سیاست زمان‌بندی قطعی تر است و به کاهش تأخیر کمک می‌کند. (real-time)
- ۶.۱.۳ پیادهسازی رابط خط فرمان (CLI) و چرخه حیات محفظه سیستم مدیریت محفظه از طریق یک رابط خط فرمان جامع کنترل می‌شود. تابع main برنامه به عنوان یک توزیع‌کننده (dispatcher) عمل کرده و با استفاده از strcmp، اجرای برنامه را به توابع * do_* مربوطه هدایت می‌کند.
- run: این دستور، نقطه‌ی شروع ایجاد یک محفظه است و تمام مراحل راهاندازی شامل ایجاد namespaces، cgroups و OverlayFS را هماهنگ می‌کند. فلگ detach-- باعث می‌شود تا پروسه‌ی والد بالافاصله پس از راهاندازی موفقیت‌آمیز محفظه، خارج شود و شل را اشغال نکند.
- status و list: این دستورات با خواندن اطلاعات از دایرکتوری‌های وضعیت در مسیر /run/my_runtime در sys/fs/cgroup، وضعیت و منابع مصرفی محفظه‌ها را نمایش می‌دهند. این رویکرد، معماری بدون دیمون سیستم را به نمایش می‌گذارد.
- rm و stop: دستور stop با ارسال سیگنال SIGKILL پروسه‌ی محفظه را خاتمه می‌دهد. دستور rm پس از اطمینان از متوقف بودن پروسه، تمام منابع مرتبط با آن، شامل دایرکتوری وضعیت و لایه‌ی قابل نوشتن OverlayFS، را از روی سیستم پاک می‌کند.
- start: این دستور یک محفظه‌ی متوقف شده را مجدداً راهاندازی می‌کند. این فرآیند شامل خواندن تمام تنظیمات ذخیره شده از دایرکتوری OverlayFS، مانت مجدد PID. و ایجاد یک پروسه‌ی کاملاً جدید با یک PID جدید است. سپس دایرکتوری وضعیت به نام PID جدید تغییر نام می‌یابد.
- thaw و freeze: این مکانیزم با stop متفاوت است. این دستورات صرفاً با نوشتن مقدار '1' یا '0' در فایل cgroup.freeze مربوط به محفوظه، تمام پروسه‌های آن را به صورت موقت متوقف می‌کنند یا از سر می‌گیرند، در حالی که وضعیت داخلی آن‌ها در حافظه به طور کامل حفظ می‌شود.

۷.۱.۳ پیادهسازی نظارت با eBPF

یک ابزار نظارتی مجزا به نام monitor.py با استفاده از فناوری eBPF پیادهسازی شده است.

1. این اسکریپت پایتون با استفاده از کتابخانه bcc، یک برنامه‌ی نوشته شده به زبان C را کامپایل کرده و در هسته‌ی لینوکس بارگذاری می‌کند.
2. برنامه‌ی eBPF خود را به دو نقطه رهگیری (tracepoint) در هسته متصل می‌کند: sys_enter_clone برای رهگیری فراخوانی ایجاد clone، پروسه، و sys_enter_mkdir برای رهگیری فراخوانی ایجاد دایرکتوری.
3. در داخل هسته، برنامه رویدادها را فیلتر می‌کند. ابتدا بررسی می‌کند که آیا نام پروسه‌ی فراخوانده my_runner است یا خیر. سپس برای mkdir، پرچم‌های مربوط به ایجاد namespace را بررسی کرده و برای mkdir، مسیر ایجاد دایرکتوری را چک می‌کند تا مطمئن شود در sys/fs/cgroup قرار دارد.
4. در صورت تطابق شرایط، یک ساختار داده با اطلاعات رویداد پر شده و از طریق یک (perf buffer) به فضای کاربری (user-space) ارسال می‌شود.
5. اسکریپت پایتون این رویدادها را از بافر خوانده و پس از قالب‌بندی، در فایل ebpf_log.txt ذخیره می‌کند.
این روش، نظارتی دقیق و با سربار بسیار کم را بدون نیاز به تغییر در کد اصلی برنامه فراهم می‌آورد.

۲.۳ بررسی کد و توابع فایل‌های اصلی

در این بخش، به بررسی دقیق‌تر کد منبع برنامه (main.c) و سپس سایر فایل‌ها می‌پردازیم. این تحلیل به درک عمیق‌تر نحوه تعامل برنامه با هسته لینوکس و پیاده‌سازی قابلیت‌های مختلف کمک می‌کند.

۱.۲.۳ تعاریف و ثابت‌های سراسری

در ابتدای برنامه، مجموعه‌ای از ثابت‌ها با استفاده از دستور پیش‌پردازنده `#define` تعریف شده‌اند.

```
#define STACK_SIZE (1024 * 1024)
#define MY_RUNTIME_CGROUP "/sys/fs/cgroup/my_runtime"
#define MY_RUNTIME_STATE "/run/my_runtime"
#define NEXT_CPU_FILE "/tmp/my_runtime_next_cpu"
```

• **STACK SIZE**: این ثابت، حجم پشتی حافظه را برای پروسه‌ی جدید محفظه تعیین می‌کند. مقدار ۱MB یک اندازه‌ی استاندارد و امن برای جلوگیری از سرریز پشتی (stack overflow) در اکثر برنامه‌ها است. این پشتی با فراخوانی `malloc` تخصیص داده شده و اشاره‌گر آن به فراخوانی سیستمی `clone()` ارسال می‌شود.

• **MY RUNTIME CGROUP**: این مسیر، دایرکتوری ریشه‌ی `cgroup` را برای این برنامه مشخص می‌کند. تمام زیرگروه‌های مربوط به هر محفظه در این مسیر ایجاد می‌شوند. استفاده از یک دایرکتوری اختصاصی، مدیریت منابع را متمرکز کرده و از تداخل با سایر سرویس‌های سیستم جلوگیری می‌کند.

• **MY RUNTIME STATE**: این مسیر، دایرکتوری را مشخص می‌کند که اطلاعات وضعیت (state) مربوط به هر محفظه در حال اجرا یا متوقف شده در آن ذخیره می‌شود. برای هر محفظه، یک زیردایرکتوری با نام `PID` آن ایجاد شده و فایل‌های حاوی اطلاعات پیکربندی (مانند نام ایمیج، دستور اجرا شده، و فلگ‌های استفاده شده) در آن قرار می‌گیرد. این رویکرد، اساس معماری بدون دیمون برنامه است.

• **NEXT CPU FILE**: این ثابت، مسیر یک فایل موقت را برای پیاده‌سازی قابلیت `CPU Pinning` به صورت `round-robin` تعریف می‌کند. شماره‌ی آخرین هسته‌ی `CPU` تخصیص داده شده در این فایل ذخیره می‌شود تا محفوظه‌ی بعدی به هسته‌ی بعدی قفل شود.

۲.۲.۴ توابع کمکی

در برنامه، تعدادی تابع کمکی برای انجام وظایف تکراری و ساده‌سازی کد اصلی نوشته شده است.

تابع `read_file_string` این تابع ابزاری برای خواندن محتوای متنی (یک خط) از یک فایل مشخص است. این تابع به طور گسترده در برنامه، به خصوص در دستور `status`، برای خواندن اطلاعات از فایل‌های وضعیت محفظه و فایل‌های رابط `cgroup` استفاده می‌شود.

```
void read_file_string(const char *path, char *buf, size_t size) {
    FILE *f = fopen(path, "r");
    if (f) {
        if (fgets(buf, size, f) != NULL) {
            buf[strcspn(buf, "\n")] = 0;
        } else {
            buf[0] = '\0';
        }
        fclose(f);
    } else {
        buf[0] = '\0';
    }
}
```

این تابع یک مسیر فایل، یک بافر برای ذخیره‌سازی رشته، و اندازه‌ی آن بافر را به عنوان ورودی دریافت می‌کند. عملکرد آن به این صورت است که ابتدا با استفاده از `fopen` را در حالت فقط خواندنی باز می‌کند. در صورت موفقیت، با استفاده از `fgets` یک خط از آن را می‌خواند. یک نکته‌ی کلیدی در این تابع، استفاده از

```
buf[strcspn(buf, "\n")] = 0;
```

است. این دستور کاراکتر (`newline`) را که به طور خودکار توسط `fgets` به انتهای رشته اضافه می‌شود، پیدا کرده و آن را با کاراکتر نال (`null terminator`) جایگزین می‌کند. این کار تضمین می‌کند که رشته‌ی خروجی تمیز و بدون کاراکترهای اضافی باشد. این تابع همچنین مدیریت خط را به خوبی انجام می‌دهد؛ اگر فایل وجود نداشته باشد یا خواندن آن با مشکل مواجه شود، بافر ورودی را به یک رشته‌ی خالی تبدیل می‌کند تا از استفاده از داده‌های نامعتبر در ادامه‌ی برنامه جلوگیری شود.

تابع cleanup_mounts این تابع مسئولیت پاکسازی و *unmount* کردن تمام فایل سیستم‌های مرتبط با یک محفظه را بر عهده دارد. فراخوانی این تابع در دستورات `rm stop` و ضروری است تا از باقی ماندن نقاط مانت سرگردان (*dangling mount points*) که می‌توانند باعث بروز مشکل در سیستم شوند، جلوگیری شود.

```
void cleanup_mounts(int pid) {
    char state_dir[PATH_MAX];
    snprintf(state_dir, sizeof(state_dir), "%s/%d", MY_RUNTIME_STATE, pid);
    char overlay_id_path[PATH_MAX];
    snprintf(overlay_id_path, sizeof(overlay_id_path), "%s/overlay_id", state_dir);
    int random_id = -1;
    FILE* id_file = fopen(overlay_id_path, "r");
    if (id_file) {
        fscanf(id_file, "%d", &random_id);
        fclose(id_file);
    }
    if (random_id != -1) {
        char merged[PATH_MAX];
        snprintf(merged, sizeof(merged), "overlay_layers/%d/merged", random_id);
        char proc_to_unmount[PATH_MAX];
        snprintf(proc_to_unmount, sizeof(proc_to_unmount), "%s/proc", merged);

        if (umount2(proc_to_unmount, MNT_DETACH) != 0) {
            if (errno != ENOENT && errno != EINVAL) {
                perror("umount2 proc failed");
            }
        }

        char propagate_mount_path[PATH_MAX];
        snprintf(propagate_mount_path, sizeof(propagate_mount_path),
                 "%s/propagate_mount_dir", state_dir);
        FILE* p_file = fopen(propagate_mount_path, "r");
        if (p_file) {
            char p_dir[PATH_MAX];
            if (fgets(p_dir, sizeof(p_dir), p_file)) {
                p_dir[strcspn(p_dir, "\n")] = 0;
                char container_mount_point[PATH_MAX];
                snprintf(container_mount_point, sizeof(container_mount_point),
                         "%s%s", merged, p_dir);
                if (umount2(container_mount_point, MNT_DETACH) != 0) {
                    if (errno != ENOENT && errno != EINVAL) {
                        perror("umount2 propagated mount failed");
                    }
                }
            }
            fclose(p_file);
        }

        if (umount2(merged, MNT_DETACH) != 0) {
            if (errno != ENOENT && errno != EINVAL) {
                perror("umount2 overlay failed");
            }
        }
    }
}
```

عملکرد این تابع به این صورت است که ابتدا با استفاده از *PID* محفوظه، به دایرکتوری وضعیت آن در مسیر `/run/my_runtime` مراجعه کرده و شناسه‌ی لایه‌ی *OverlayFS* آن را از فایل `overlay_id` می‌خواند. سپس، مسیر دایرکتوری ادغامشده `merged` را بازسازی می‌کند. یک نکته‌ی کلیدی در این تابع، ترتیب عملیات `umount` است. ابتدا نقاط ماند `proc` و مانت‌های اشتراکی (*propagated mounts*) باید `umount` شوند و در نهایت، خود فایل سیستم اصلی *OverlayFS* که در مسیر `merged` قرار دارد، `umount` می‌شود. این ترتیب برای جلوگیری از بروز خطای "دستگاه مشغول است" (*device is busy*) ضروری است. در تمام فراخوانی‌ها از `(2)` به همراه پرچم `MNT_DETACH` استفاده شده است. این پرچم یک عملیات `umount` "تبیل" (*lazy*) را انجام می‌دهد که حتی اگر نقطه‌ی مانت در حال استفاده باشد، بلا فاصله بازمی‌گردد و هسته در اولین فرصت ممکن آن را جدا می‌کند. این رویکرد، پایداری عملیات پاکسازی را به شکل قابل توجهی افزایش می‌دهد.

تابع write_file این تابع یک ابزار ساده و پرکاربرد برای نوشتن یک رشته‌ی متنی در یک فایل است. وظیفه‌ی اصلی آن، کپسوله کردن عملیات باز کردن، نوشتن و بستن فایل است تا از تکرار کد در بخش‌های مختلف برنامه جلوگیری شود. این تابع به طور گسترده برای ایجاد و بهروزرسانی فایل‌های وضعیت در دایرکتوری `/run/my_runtime` و همچنین برای پیکربندی فایل‌های رابط `cgroups` استفاده می‌شود.

```
void write_file(const char *path, const char *content) {
    FILE *f = fopen(path, "w");
    if (f == NULL) {
        fprintf(stderr, "ERROR: Failed to open %s for writing: ", path);
        perror("");
        return;
    }
    fprintf(f, "%s", content);
    fclose(f);
}
```

این تابع با استفاده از `fopen` با حالت `"w"` فایل را برای نوشتن باز می‌کند. این حالت به طور خودکار فایل را در صورت عدم وجود ایجاد کرده و در صورت وجود، محتوای قبلی آن را پاک می‌کند. سپس با `fprintf` محتوای ورودی را در فایل می‌نویسد و در نهایت با `fclose` آن را می‌بندد. این تابع همچنین شامل یک بررسی خطای زمانی است که باز کردن فایل با مشکل مواجه شود.

تابع setup_cgroup_hierarchy این تابع وظیفه‌ی آماده‌سازی زیرساخت‌های لازم برای مدیریت `cgroup`‌ها و ذخیره‌سازی وضعیت محفظه‌ها را بر عهده دارد. این تابع در ابتدای اجرای دستور `run` فراخوانی می‌شود تا اطمینان حاصل شود که دایرکتوری‌های مورد نیاز سیستم وجود دارند.

```
void setup_cgroup_hierarchy() {
    mkdir(MY_RUNTIME_CGROUP, 0755);
    mkdir(MY_RUNTIME_STATE, 0755);
    system("echo "+cpu+memory+pids+io" > /sys/fs/cgroup/my_runtime/cgroup.subtree_control 2>/dev/null || true");
}
```

این تابع ابتدا با استفاده از `mkdir`، دایرکتوری اصلی برنامه را در مسیر `cgroups` می‌سازد، دایرکتوری اصلی وضعیت را در مسیر `/run` ایجاد می‌کند. مهم‌ترین بخش این تابع، اجرای یک دستور سیستمی با فراخوانی `system()` است. این دستور، کنترلرهای منابع مورد نیاز (شامل `cpu`, `memory`, `pids`, و `io`) را در فایل `cgroup.subtree_control` اصلی برنامه فعال می‌کند. این عمل در `cgroups v2` یک پیش‌نیاز ضروری است و به برنامه‌ی ما اجازه می‌دهد تا این کنترلرهای زیرگروه‌هایی که برای هر محفظه ایجاد می‌کند، مدیریت نماید. عبارت `true || 2>/dev/null` برای جلوگیری از خطأ در صورتی که کنترلرهای از قبل فعال شده باشند، استفاده شده است.

ساختار container_args این ساختار داده (`struct`) به عنوان یک مکانیزم برای انتقال اطلاعات از پروسه‌ی والد (برنامه‌ی `clone()` به پروسه‌ی فرزند) که قرار است به محفظه تبدیل شود، طراحی شده است. یک اشاره‌گر به این ساختار به عنوان آرگومان به فراخوانی سیستمی (`clone()`) ارسال می‌شود.

```
struct container_args {
    char* merged_path;
    char** argv;
    char* propagate_mount_dir;
    int sync_pipe_read_fd;
};
```

اعضای این ساختار عبارتند از: `merged_path` که مسیر دایرکتوری ادغام‌شده‌ی `OverlayFS` است و به عنوان ریشه‌ی فایل سیستم محفظه استفاده خواهد شد؛ `argv` که یک آرایه از رشته‌ها برای دستوری است که محفظه باید اجرا کند؛ `propagate_mount_dir` که مسیر دایرکتوری مورد نظر برای قابلیت انتشار مانند را نگه می‌دارد؛ و در نهایت `sync_pipe_read_fd` که توصیف‌گر فایل (`file descriptor`) سمت خواندنی (`pipe`) همگام‌سازی است و برای حل مشکل `User Namespace` ایجاد `race condition` در هنگام ایجاد `User Namespace` به کار می‌رود.

تابع container_main این تابع، نقطه‌ی ورود (`entry point`) برای پروسه‌ی فرزندی است که توسط `clone()` ایجاد می‌شود. تمام کدی که در این تابع قرار دارد، در داخل `namespace`‌های جدید و ایزوله اجرا می‌شود. وظیفه‌ی اصلی این تابع، آماده‌سازی نهایی محیط محفظه و سپس جایگزین کردن پروسه‌ی فعلی با دستوری است که کاربر قصد اجرای آن را داشته است.

```
int container_main(void *arg) {
    struct container_args* args = (struct container_args*)arg;

    char buf;
    // Wait for the parent to finish setting up user mappings.
    if (read(args->sync_pipe_read_fd, &buf, 1) != 0) {
        perror("Child failed to sync with parent");
        return 1;
    }
    close(args->sync_pipe_read_fd);
```

```

// Bring up the loopback interface.
if (system("ip link set lo up") != 0) {
    perror("Failed to set lo up");
}

// Handle propagated mount if requested.
if (args->propagate_mount_dir) {
    // ... (mount logic)
}

sethostname("container", 9);
if (chroot(args->merged_path) != 0) { perror("chroot failed"); return 1; }
if (chdir("/") != 0) { perror("chdir failed"); return 1; }
if (mount("proc", "/proc", "proc", 0, NULL) != 0) {
    perror("mount proc failed");
}

execv(args->argv[0], args->argv);

// This part is only reached if execv fails.
perror("execv failed");
return 1;
}

```

روند اجرای این تابع به شرح زیر است. اولین و مهم‌ترین گام، همگام‌سازی با پروسه‌ی والد است. فراخوانی `read` پروسه‌ی فرزند را تا زمانی که والد نگاشته‌ای *User Namespace* را تکمیل نکرده و `pipe` را نبسته، متوقف می‌کند. این کار از بروز *race condition* دستورات با مجوزهای نادرست جلوگیری می‌کند. پس از همگام‌سازی، تابع با اجرای دستور `ip link set lo up`، رابط شبکه‌ی محلی را در *Network Namespace* جدید فعال می‌کند. سپس، در صورت درخواست کاربر، نقطه‌ی مانت آشنا کی را پیکربندی می‌کند. در مرحله‌ی بعد، با استفاده از فراخوانی‌های سیستمی، محیط محفظه نهایی می‌شود. `UTS Namespace` نام میزبان را در `sethostname` پرسه را در فایل سیستم مجازی *OverlayFS* محصور می‌کند. پس از آن، فایل سیستم مجازی `proc` در داخل محفظه مانت می‌شود. نقطه‌ی پایانی و بدون بازگشت این تابع، فراخوانی `execv` است. این فراخوانی، ایمیج پروسه‌ی فعلی (یعنی برنامه‌ی `my_runner`) را با ایمیج دستوری که کاربر مشخص کرده است، جایگزین می‌کند. به عبارت دیگر، از این نقطه به بعد، دیگر کد ما اجرا نمی‌شود و پرسه به طور کامل به برنامه‌ی مورد نظر کاربر تبدیل می‌شود. به همین دلیل، هر کدی که پس از `execv` قرار دارد (مانند `perror`) تنها در صورتی اجرا خواهد شد که فراخوانی `execv` با شکست مواجه شود.

تابع `read_cgroup_long` این تابع برای خواندن یک مقدار عددی از نوع `long` از یک فایل مشخص در فایل سیستم مجازی *cgroups* طراحی شده است. بسیاری از فایل‌های رابط `cgroups`، مانند `memory.current` یا `pids.current`، تنها حاوی یک مقدار عددی هستند. این تابع این فرآیند را ساده‌سازی می‌کند.

```

long read_cgroup_long(const char *path) {
    long value = -1;
    FILE *f = fopen(path, "r");
    if (!f) return -1;
    if (fscanf(f, "%ld", &value) != 1) { value = -1; }
    fclose(f);
    return value;
}

```

تابع مسیر فایل را به عنوان ورودی دریافت کرده و با استفاده از `fopen` و `fscanf` آن را باز می‌کند. سپس با `%ld` و فرمت دهنده `value` را می‌خواند و در متغیر `value` ذخیره می‌کند. این تابع همچنین شامل بررسی خطای است: اگر فایل باز نشود یا خواندن عدد با موفقیت انجام نشود (مثلاً فایل خالی باشد)، مقدار پیش‌فرض ۱ – را بازمی‌گرداند تا نشان‌دهنده‌ی عدم موفقیت باشد.

تابع `format_bytes` این تابع یک ابزار کاربردی برای تبدیل یک مقدار عددی بزرگ (که معمولاً تعداد بایت‌ها را نشان می‌دهد) به یک رشته‌ی خوانا برای انسان است. به جای نمایش یک عدد طولانی مانند 1048576، این تابع آن را به صورت ۰.۱ MB ۰۰.۰ نمایش می‌دهد. این تابع به طور گسترده در دستور `status` برای نمایش مصرف حافظه و آمار ورودی/خروجی استفاده می‌شود.

```

void format_bytes(long bytes, char *buf, size_t size) {
    const char* suffixes[] = {"B", "KB", "MB", "GB", "TB"};
    int i = 0;
    double d_bytes = bytes;
    if (bytes < 0) {
        snprintf(buf, size, "N/A");
    }
    while (d_bytes > 1024 && i < 4) {
        d_bytes /= 1024;
        i++;
    }
    if (i == 4) {
        snprintf(buf, size, "%.1f TB", d_bytes);
    } else {
        snprintf(buf, size, "%.1f %s", d_bytes, suffixes[i]);
    }
}

```

```

        return;
    }
    while (d_bytes >= 1024 && i < 4) {
        d_bytes /= 1024;
        i++;
    }
    snprintf(buf, size, "%.2f %s", d_bytes, suffixes[i]);
}

```

این تابع یک مقدار عددی از نوع `long` را به عنوان ورودی می‌گیرد و با یک حلقه‌ی `while`, به طور مکرر آن را بر 10^{24} تقسیم می‌کند تا به واحد مناسب (بايت، کیلوبایت، مگابایت و غیره) برسد. در هر تکرار، یک شمارنده افزایش می‌باید تا پسوند صحیح از آرایه‌ی `suffixes` انتخاب شود. در نهایت، با استفاده از `snprintf`, مقدار نهایی با دو رقم اعشار به همراه پسوند مناسب در بافر خروجی فرمت‌بندی می‌شود.

تابع `find_cgroup_value` این تابع برای خواندن یک مقدار عددی خاص از فایل‌هایی طراحی شده است که حاوی چندین جفت کلید-مقدار هستند، مانند فایل `io.stat` یا `cpu.stat` در `cgroups`. این تابع به ما اجازه می‌دهد تا تنها مقدار مورد نظر خود را بر اساس کلید آن استخراج کنیم.

```

long find_cgroup_value(const char* path, const char* key) {
    FILE* f = fopen(path, "r");
    if (!f) return -1;
    char line_buf[256];
    long value = -1;
    while (fgets(line_buf, sizeof(line_buf), f) != NULL) {
        char key_buf[128];
        long val_buf;
        if (sscanf(line_buf, "%s %ld", key_buf, &val_buf) == 2) {
            if (strcmp(key_buf, key) == 0) {
                value = val_buf;
                break;
            }
        }
    }
    fclose(f);
    return value;
}

```

این تابع مسیر فایل و کلید مورد نظر را به عنوان ورودی دریافت می‌کند. سپس با استفاده از `fgets`, فایل را خط به خط می‌خواند. در هر خط، با `sscanf` تلاش می‌کند تا یک جفت کلید (رشته) و مقدار (عدد) را استخراج کند. اگر این کار موفقیت‌آمیز بود، کلید استخراج شده را با کلید ورودی مقایسه می‌کند. در صورت تطابق، مقدار عددی متناظر را ذخیره کرده و با دستور `break` از حلقه خارج می‌شود تا از پردازش اضافی جلوگیری شود.

۳.۲.۳ `do_run` تابع

این تابع، پیچیده‌ترین و در عین حال اصلی‌ترین بخش برنامه است که مسئولیت کامل ایجاد و راهاندازی یک محفظه‌ی جدید را بر عهده دارد. این تابع تمام مراحل لازم، از تجزیه‌ی آرگومان‌های ورودی گرفته تا ایجاد `(namespace)`‌ها و پیکربندی `(cgroup)`‌ها را هماهنگ می‌کند. به دلیل طولانی بودن، این تابع را به چند بخش منطقی تقسیم کرده و هر بخش را به صورت جداگانه بررسی می‌کنیم.

بخش اول: مقداردهی اولیه و تجزیه آرگومان‌ها در ابتدای تابع، متغیرهای محلی برای نگهداری مقادیر فلگ‌های خط فرمان تعریف می‌شوند. سپس، با استفاده از تابع `getopt_long`, آرگومان‌های ورودی کاربر تجزیه (parse) می‌شوند. این تابع به ما اجازه می‌دهد تا هم فلگ‌های کوتاه (`-d`) و هم فلگ‌های بلند (مانند `--detach`) را به راحتی مدیریت کنیم.

```

int do_run(int argc, char *argv[]) {
    setup_cgroup_hierarchy();
    char *mem_limit = NULL;
    char *cpu_quota = NULL;
    // ... (other variable declarations)
    int detach_flag = 0;
    int share_ipc_flag = 0;

    static struct option long_options[] = {
        {"mem", required_argument, 0, 'm'},
        {"cpu", required_argument, 0, 'C'},
        // ... (all other options)
}

```

```

    {"propagate-mount", required_argument, 0, 'M'},
    {0, 0, 0, 0}
};

int opt;
while ((opt = getopt_long(argc, argv, "+m:C:r:w:pdiM:",
    long_options, NULL)) != -1) {
    switch (opt) {
        case 'm': mem_limit = optarg; break;
        case 'd': detach_flag = 1; break;
        // ... (cases for all other flags)
        default: return 1;
    }
}

if (optind + 1 >= argc) { /* error handling */ }

char* image_name = argv[optind];
char** container_cmd_argv = &argv[optind + 1];
// ...

```

یک حلقه‌ی while تمام فلگ‌ها را یکی‌یکی پردازش کرده و مقادیر مربوطه را در متغیرها ذخیره می‌کند. پس از پایان حلقه، متغیر optind اندیس اولین آرگومان غیرفلگ را در خود دارد که همان نام ایمیج پایه است. آرگومان‌های بعدی نیز به عنوان دستوری که باید در محفظه اجرا شود، در نظر گرفته می‌شوند.

بخش دوم: آماده‌سازی فایل‌سیستم OverlayFS در این بخش، زیرساخت فایل‌سیستم مجازی برای محفظه آماده می‌شود. ابتدا، در صورت نیاز، قابلیت انتشار مانت پیکربندی می‌شود. سپس، لایه‌های لازم برای OverlayFS ایجاد و درنهایت با یکدیگر ادغام می‌شوند.

```

// ... (code for propagate_mount_dir if set)

char lowerdir[PATH_MAX], upperdir[PATH_MAX], workdir[PATH_MAX], merged[PATH_MAX];
srand(time(NULL) ^ getpid());
int random_id = rand() % 10000;
snprintf(lowerdir, sizeof(lowerdir), "%s", image_name);
snprintf(upperdir, sizeof(upperdir), "overlay_layers/%d/upper", random_id);
snprintf(workdir, sizeof(workdir), "overlay_layers/%d/work", random_id);
snprintf(merged, sizeof(merged), "overlay_layers/%d/merged", random_id);

char command[PATH_MAX * 2];
sprintf(command, "mkdir -p %s %s %s", upperdir, workdir, merged);
if (system(command) != 0) { return 1; }

char mount_opts[PATH_MAX * 3];
snprintf(mount_opts, sizeof(mount_opts), "lowerdir=%s,upperdir=%s,workdir=%s",
    lowerdir, upperdir, workdir);
if (mount("overlay", merged, "overlay", 0, mount_opts) != 0) {
    perror("Overlay mount failed");
    return 1;
}

```

برای هر محفظه، یک شناسه‌ی عددی تصادفی ایجاد می‌شود تا نام دایرکتوری لایه‌ی قابل نوشتن آن منحصر به فرد باشد. سپس مسیرهای (ایمیج پایه)، upperdir (لایه‌ی قابل نوشتن)، workdir (دایرکتوری کاری) و merged (نقشه‌ی مانت نهایی) با استفاده از snprintf ساخته می‌شوند. پس از ایجاد این دایرکتوری‌ها با دستور pmkdir، گزینه‌های مانت در یک رشته فرمت‌بندی شده و درنهایت، فراخوانی سیستمی () با نوع overlay اجرا می‌شود تا فایل‌سیستم مجازی ایجاد گردد.

بخش سوم: ایجاد پروسه محفظه و همگام‌سازی این بخش، هسته‌ی اصلی ایجاد محفظه است. در اینجا، پروسه‌ی فرزند با استفاده از () ایجاد شده و مکانیزم همگام‌سازی بین والد و فرزند پیاده‌سازی می‌شود.

```

int sync_pipe[2];
if (pipe(sync_pipe) == -1) { /* error handling */ }

struct container_args args;
args.merged_path = merged;
args.argv = container_cmd_argv;
args.propagate_mount_dir = propagate_mount_dir;
args.sync_pipe_read_fd = sync_pipe[0];

```

```

char *container_stack = malloc(STACK_SIZE);
char *stack_top = container_stack + STACK_SIZE;

int clone_flags = CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWUTS |
                  CLONE_NEWUSER | CLONE_NEWNET | SIGCHLD;
if (!share_ipc_flag) {
    clone_flags |= CLONE_NEWIIPC;
}
pid_t container_pid = clone(container_main, stack_top, clone_flags, &args);

if (container_pid == -1) { /* error handling */ }

close(sync_pipe[0]); // Parent closes the read end of the pipe

```

ابتدا یک (pipe) برای همگامسازی ایجاد می‌شود. سپس، یک نمونه از ساختار container_args با اطلاعات لازم (مانند مسیر دایرکتوری merged و آرگومان‌های دستور) پر می‌شود. پشتیهی حافظهی مورد نیاز برای پروسه‌ی جدید تخصیص داده می‌شود. سپس، پرچم‌های لازم برای ایجاد تمام namespace‌ها در متغیر clone_flags جمع‌آوری می‌شوند. در نهایت، فراخوانی سیستمی () clone با مشخص کردن تابع container_main به عنوان نقطه‌ی شروع پروسه‌ی فرزند، پشتیهی حافظه، پرچم‌ها و آرگومان‌ها اجرا می‌شود. پس از ایجاد موفقیت‌آمیز فرزند، پروسه‌ی والد بلافاصله سمت خواندنی (pipe) را می‌بندد، زیرا دیگر به آن نیازی ندارد.

بخش چهارم: پیکربندی User Namespace و ارسال سیگنال بلافاصله پس از ایجاد پروسه‌ی فرزند، والد باید نگاشت کاربران را برای User Namespace جدید پیکربندی کند.

```

// ... (logic to get host_uid and host_gid from getenv or getuid)

snprintf(path_buffer, sizeof(path_buffer), "/proc/%ld/setgroups",
(long)container_pid);
write_file(path_buffer, "deny");

snprintf(path_buffer, sizeof(path_buffer), "/proc/%ld/gid_map",
(long)container_pid);
char map_buffer[100];
snprintf(map_buffer, sizeof(map_buffer), "0 %d 1", host_gid);
write_file(path_buffer, map_buffer);

snprintf(path_buffer, sizeof(path_buffer), "/proc/%ld/uid_map",
(long)container_pid);
snprintf(map_buffer, sizeof(map_buffer), "0 %d 1", host_uid);
write_file(path_buffer, map_buffer);

// Signal the child that mapping is done
if (write(sync_pipe[1], "1", 1) != 1) {
    perror("write to sync pipe");
}
close(sync_pipe[1]);

```

در این بخش، ابتدا شناسه‌های کاربر و گروه واقعی از طریق متغیرهای محیطی SUDO یا با فراخوانی getuid در مسیر /proc/[pid]/uid_map و /proc/[pid]/gid_map، setgroups می‌نویسد و سپس آن را می‌بندد. این عمل، سیگنالی برای فرزند است که از حالت انتظار خارج شده و کار خود را ادامه دهد.

بخش پنجم: ذخیره‌سازی وضعیت و اعمال محدودیت‌ها در این بخش، تمام اطلاعات پیکربندی و محدودیت‌های متابع برای محفظه در فایل سیستم ذخیره و اعمال می‌شوند.

```

char state_dir[PATH_MAX];
snprintf(state_dir, sizeof(state_dir), "%s/%ld",
         MY_RUNTIME_STATE, (long)container_pid);
mkdir(state_dir, 0755);

char cgroup_path[PATH_MAX];
snprintf(cgroup_path, sizeof(cgroup_path), "%s/container_%ld",
         MY_RUNTIME_CGROUP, (long)container_pid);
mkdir(cgroup_path, 0755);

```

```

// ... (writing command, image_name, overlay_id, and flags to state_dir)

if (mem_limit) {
    // ... (writing mem_limit to state_dir and memory.max to cgroup_path)
}
// ... (logic for other resource limits like cpu, io)

char procs_path[PATH_MAX];
snprintf(procs_path, sizeof(procs_path), "%s/cgroup.procs",
cgroup_path);
snprintf(pid_str, sizeof(pid_str), "%ld", (long)container_pid);
write_file(procs_path, pid_str);

```

ابتدا دایرکتوری وضعیت و دایرکتوری *cgroup* برای محفظه‌ی جدید ایجاد می‌شوند. سپس، تمام اطلاعات پیکربندی، از جمله نام ایمیج، دستور اجرا، و فلگ‌های استفاده شده مانند *--detach*، در فایل‌های متنی جداگانه در داخل دایرکتوری وضعیت نوشته می‌شوند. این کار به دستورات بعدی مانند *status* و *start* اجازه می‌دهد تا تنظیمات اولیه را بازیابی کنند. هم‌زمان، محدودیت‌های منابع درخواست شده (*CPU*) در فایل‌های مربوطه در دایرکتوری *cgroup* نوشته می‌شوند. در نهایت، *PID* محفظه در فایل *cgroup.procs* نوشته می‌شود تا تمام این قوانین بر روی آن اعمال گردد.

بخش ششم: نهایی‌سازی و مدیریت حالت اجرایی این بخش پایانی، رفتار برنامه را بر اساس حالت اتصال (*attached*) یا جدا شده (*detached*) تعیین می‌کند.

```

if (detach_flag) {
    printf("Container started with PID %ld\n", (long)container_pid);
    return 0;
}

printf("Container started with PID %ld. Press Ctrl+C to stop.\n",
       (long)container_pid);
waitpid(container_pid, NULL, 0);
printf("Container %ld has exited. Use 'rm' to clean up.\n",
       (long)container_pid);
return 0;

```

اگر فلگ *--detach* توسط کاربر مشخص شده باشد، برنامه صرفاً *PID* محفظه را چاپ کرده و بلا فاصله خارج می‌شود. این کار به محفظه اجازه می‌دهد تا در پس زمینه به کار خود ادامه دهد. اما اگر این فلگ استفاده نشده باشد (حالت پیش‌فرض)، برنامه با استفاده از *fork* و *execve* منتظر می‌ماند تا پروسه‌ی محفظه به پایان برسد. این به کاربر اجازه می‌دهد تا خروجی استاندارد محفظه را به صورت زنده مشاهده کرده و با آن تعامل داشته باشد.

۴.۲.۳ تابع *do_list*

این تابع، که مسئولیت اجرای دستور *list* را بر عهده دارد، یک نمونه‌ی بارز از پیاده‌سازی معماری بدون دیمون است. این تابع برای به دست آوردن لیست محفظه‌ها و وضعیت آن‌ها، به هیچ پروسه‌ی پس‌زمینه‌ی دائمی متصل نمی‌شود، بلکه تمام اطلاعات مورد نیاز خود را مستقیماً از فایل سیستم می‌خواند.

```

int do_list(int argc, char *argv[]) {
    DIR *d = opendir(MY_RUNTIME_STATE);
    if (d == NULL) {
        if (errno == ENOENT) {
            printf("No containers exist.\n");
            return 0;
        }
        perror("opendir");
        return 1;
    }

    struct dirent *dir_entry;
    int found = 0;

    while ((dir_entry = readdir(d)) != NULL) {
        if (dir_entry->d_type != DT_DIR || strcmp(dir_entry->d_name, ".") == 0 ||
            strcmp(dir_entry->d_name, "..") == 0)
            continue;

        if (!found) {

```

```

        printf("%-15s\t%-10s\t%s\n", "CONTAINER PID", "STATUS", "COMMAND");
        found = 1;
    }

    char proc_path[PATH_MAX];
    snprintf(proc_path, sizeof(proc_path), "/proc/%s", dir_entry->d_name);
    const char* status = (access(proc_path, F_OK) == 0) ? "Running" : "Stopped";

    char cmd_path[PATH_MAX], cmd_buf[1024] = {0};
    snprintf(cmd_path, sizeof(cmd_path), "%s/%s/command",
             MY_RUNTIME_STATE, dir_entry->d_name);

    // Using our helper function to read the command
    read_file_string(cmd_path, cmd_buf, sizeof(cmd_buf));

    printf("%-15s\t%-10s\t%s\n", dir_entry->d_name, status, cmd_buf);
}

closedir(d);

if (!found) {
    printf("No containers exist.\n");
}

return 0;
}

```

روند اجرای این تابع به این صورت است که ابتدا با استفاده از `opendir`، دایرکتوری وضعیت اصلی برنامه (`/run/my_runtime`) را باز می‌کند. سپس، در یک حلقه `while` و با استفاده از `readdir`، تمام ورودی‌های داخل این دایرکتوری را یکی‌یکی پیمایش می‌کند. از آنجایی که هر محفظه‌ی فعال یا متوقف شده یک زیردایرکتوری با نام `PID` خود در این مسیر دارد، نام هر دایرکتوری به عنوان شناسه‌ی یک محفظه در نظر گرفته می‌شود. برای تعیین وضعیت محفظه (در حال اجرا یا متوقف)، تابع با استفاده از فراخوانی `access`، وجود دایرکتوری منتظر با آن `PID` را در فایل سیستم مجازی `/proc` بررسی می‌کند. اگر دایرکتوری وجود داشته باشد، به معنای "Running" بودن محفظه است و در غیر این صورت، "Stopped" در نظر گرفته می‌شود. در نهایت، دستور اجرایی محفظه از فایل `command` داخل دایرکتوری وضعیت خوانده شده و تمام اطلاعات در یک جدول با فرمت‌بندی مناسب چاپ می‌شوند. یک پرچم به نام `found` نیز استفاده شده است تا سرآیند جدول تنها یک بار و فقط در صورتی که حداقل یک محفظه وجود داشته باشد، چاپ شود.

۵.۲.۳ do_status تابع

این تابع مسئولیت اجرای دستور `status` را بر عهده دارد و اطلاعات دقیقی در مورد وضعیت و منابع مصرفی یک محفظه‌ی خاص را نمایش می‌دهد. این تابع نیز مانند `do_list`، با خواندن اطلاعات از دایرکتوری وضعیت و دایرکتوری `cgroup` محفظه، کار می‌کند و به هیچ پروسه‌ی پس‌زمینه‌ای وابسته نیست.

```

int do_status(int argc, char *argv[]) {
    if (argc < 2) { /* error handling */ }
    char *pid_str = argv[1];
    char path_buffer[PATH_MAX], format_buffer[64];
    char state_dir[PATH_MAX];
    snprintf(state_dir, sizeof(state_dir), "%s/%s", MY_RUNTIME_STATE, pid_str);

    if (access(state_dir, F_OK) != 0) { /* error handling */ }

    printf("--- Status for Container PID %s ---\n", pid_str);

    snprintf(path_buffer, sizeof(path_buffer), "%s/command", state_dir);
    FILE *f = fopen(path_buffer, "r");
    if (f) {
        char cmd_buf[1024] = {0};
        fgets(cmd_buf, sizeof(cmd_buf)-1, f);
        cmd_buf[strcspn(cmd_buf, "\n")] = 0;
        printf("%-25s: %s\n", "Command", cmd_buf);
        fclose(f);
    }

    // ... (code to read and print propagate_mount_dir if it exists)
}

```

```

char cgroup_path[PATH_MAX];
snprintf(cgroup_path, sizeof(cgroup_path), "%s/container_%s",
         MY_RUNTIME_CGROUP, pid_str);
printf("\n--- Resources ---\n");

snprintf(path_buffer, sizeof(path_buffer), "%s/memory.current", cgroup_path);
long mem_current = read_cgroup_long(path_buffer);
format_bytes(mem_current, format_buffer, sizeof(format_buffer));
printf("%-25s: %s\n", "Memory Usage", format_buffer);

snprintf(path_buffer, sizeof(path_buffer), "%s/cpu.stat", cgroup_path);
long cpu_micros = find_cgroup_value(path_buffer, "usage_usec");
if (cpu_micros >= 0) {
    printf("%-25s: %.2f seconds\n", "Total CPU Time",
           (double)cpu_micros / 1000000.0);
}

snprintf(path_buffer, sizeof(path_buffer), "%s/pids.current", cgroup_path);
long pids_current = read_cgroup_long(path_buffer);
printf("%-25s: %ld\n", "Active Processes/Threads", pids_current);

printf("\n-----\n");
return 0;
}

```

روند اجرای این تابع به این صورت است که ابتدا *PID* محفظه را از آرگومان‌های خط فرمان دریافت می‌کند. سپس، با استفاده از فراخوانی *access*، وجود دایرکتوری وضعیت مربوط به آن *PID* را بررسی می‌کند تا از معنیر بودن محفظه اطمینان حاصل کند. پس از آن، اطلاعات پیکربندی اولیه مانند دستور اجرایی و مسیر مانع اشتراکی (در صورت وجود) را از فایل‌های متنی داخل دایرکتوری وضعیت می‌خواند و چاپ می‌کند. بخش اصلی این تابع، خواندن آمار مانع از دایرکتوری *cgroup* محفوظه است. این تابع با استفاده از توابع *find_cgroup_value* که قبلاً بررسی شدند (*read_cgroup_long*، *format_bytes* و *find_cgroup_value*)، مقادیر مربوط به مصرف حافظه، زمان کل پردازنده و تعداد پروسه‌های فعال را از فایل‌های *pids.current*، *cpu.stat* و *memory.current* استخراج کرده و پس از فرمت‌بندی مناسب، آنها را در خروجی نمایش می‌دهد. این رویکرد یک نمای کلی و مفید از وضعیت لحظه‌ای محفوظه را در اختیار کاربر قرار می‌دهد.

۶.۲.۳ توابع *do_thaw* و *do_freeze*

این دو تابع، مسئولیت پیاده‌سازی قابلیت توقف موقت (*suspend*) و از سرگیری (*resume*) را بر عهده دارند. این قابلیت به طور کامل با استفاده از کنترلر *freezer* در *cgroups v2* پیاده‌سازی شده است. این تابع نمونه‌ای عالی از سادگی و قدرت رابط فایل‌محور *cgroups* هستند.

```

int do_freeze(int argc, char *argv[]) {
    if (argc < 2) { /* error handling */ }
    char *pid_str = argv[1];
    char freeze_path[PATH_MAX];
    snprintf(freeze_path, sizeof(freeze_path), "%s/container_%s/cgroup.freeze",
             MY_RUNTIME_CGROUP, pid_str);
    write_file(freeze_path, "1");
    printf("Frozen container %s.\n", pid_str);
    return 0;
}

int do_thaw(int argc, char *argv[]) {
    if (argc < 2) { /* error handling */ }
    char *pid_str = argv[1];
    char freeze_path[PATH_MAX];
    snprintf(freeze_path, sizeof(freeze_path), "%s/container_%s/cgroup.freeze",
             MY_RUNTIME_CGROUP, pid_str);
    write_file(freeze_path, "0");
    printf("Thawed container %s.\n", pid_str);
    return 0;
}

```

هر دو تابع ابتدا *PID* محفوظه‌ی مورد نظر را از آرگومان‌های خط فرمان دریافت می‌کند. سپس با استفاده از *snprintf*، مسیر کامل فایل‌کنترلی *cgroup.freeze* را در دایرکتوری *cgroup* مخصوص آن محفوظه می‌سازند. تابع *do_freeze* با استفاده از تابع *kmk* (*write_file*)

مقدار رشته‌ای "۱" را در این فایل می‌نویسد. این عمل به هسته‌ی لینوکس دستور می‌دهد تا تمام پروسه‌های موجود در آن `cgroup` را در حالت توقف موقت قرار دهد. به طور مشابه، تابع `do_thaw` مقدار "۰" را در همان فایل می‌نویسد تا هسته، اجرای آن پروسه‌ها را دقیقاً از همان نقطه‌ای که متوقف شده بودند، از سر بگیرد. این پیاده‌سازی ساده، قدرت کنترل `freezer` را برای مدیریت وضعیت پیشرفتی محفظه‌ها به نمایش می‌گذارد.

۷.۲.۳ تابع `do_stop`

این تابع مسئولیت خاتمه دادن کامل به یک پروسه محفظه را بر عهده دارد. برخلاف `freeze` که پروسه را در حافظه نگه می‌دارد، `stop` پروسه را به طور کامل از بین می‌برد.

```
int do_stop(int argc, char *argv[]) {
    if (argc < 2) { /* error handling */
    pid_t pid = atoi(argv[1]);
    printf("Stopping container %d...\n", pid);
    if (kill(pid, SIGKILL) != 0) {
        perror("kill failed");
    } else {
        waitpid(pid, NULL, 0);
        cleanup_mounts(pid);
    }
    printf("Container %d stopped.\n", pid);
    return 0;
}
```

این تابع پس از دریافت *PID* محفظه، از فراخوانی سیستمی `kill` برای ارسال سیگنال `SIGKILL` به آن پروسه استفاده می‌کند. یک سیگنال غیرقابل چشمپوشی است که به هسته دستور می‌دهد تا پروسه مورد نظر را بلا فاصله و بدون هیچ‌گونه فرصتی برای پاک‌سازی داخلی، خاتمه دهد. پس از ارسال موقعیت‌آمیز سیگنال، تابع با استفاده از `waitpid` منتظر می‌ماند تا پروسه فرزند به طور کامل از جدول پروسه‌های سیستم خارج شود. این کار برای جلوگیری از ایجاد پروسه‌های زامبی (*zombie process*) ضروری است. در نهایت، تابع `cleanup_mounts` از جدول پروسه‌های فراخوانی می‌شود تا تمام نقاط مرتبط با محفظه (مانند `OverlayFS` و `proc`) به درستی (`umount`) شوند و منابع سیستم آزاد گردند.

۸.۲.۳ تابع `do_start`

این تابع مسئولیت راهاندازی مجدد یک محفظه موقوف شده را بر عهده دارد. برخلاف `do_run` که همه چیز را از ابتدا ایجاد می‌کند، باید وضعیت و پیکربندی محفظه‌ی قبلی را بازخوانی کرده و یک پروسه جدید با همان مشخصات ایجاد کند. این فرآیند شامل بازسازی فایل سیستم، تجزیه‌ی مجدد دستور، و اعمال مجدد محدودیت‌های منابع است.

بخش اول: اعتبارسنجی و بازخوانی وضعیت در ابتدای تابع، بررسی‌های اولیه برای اطمینان از امکان راهاندازی مجدد محفظه انجام می‌شود. سپس، تمام اطلاعات پیکربندی که در زمان اجرای اولیه ذخیره شده بودند، از دایرکتوری وضعیت خوانده می‌شوند.

```
int do_start(int argc, char *argv[]) {
    if (argc < 2) { /* error handling */
    char *pid_str = argv[1];
    char path_buffer[PATH_MAX];

    // Ensure container is not already running
    char proc_path[PATH_MAX];
    snprintf(proc_path, sizeof(proc_path), "/proc/%s", pid_str);
    if (access(proc_path, F_OK) == 0) { /* error handling */

    // Ensure state directory for the stopped container exists
    char old_state_dir[PATH_MAX];
    snprintf(old_state_dir, sizeof(old_state_dir), "%s/%s",
    MY_RUNTIME_STATE, pid_str);
    if (access(old_state_dir, F_OK) != 0) { /* error handling */

    printf("Starting container %s...\n", pid_str);

    // Read all configuration from the state files
    char image_name[PATH_MAX] = {0};
    char overlay_id[16] = {0};
    // ... (declarations for all other config variables)
    int original_detach_flag = 0;

    snprintf(path_buffer, sizeof(path_buffer), "%s/image_name",
    old_state_dir);
```

```

read_file_string(path_buffer, image_name, sizeof(image_name));

snprintf(path_buffer, sizeof(path_buffer), "%s/overlay_id",
old_state_dir);
read_file_string(path_buffer, overlay_id, sizeof(overlay_id));

// ... (reading all other config files like command, detach,
mem_limit, etc.)

if (strlen(image_name) == 0 || strlen(overlay_id) == 0 ||
    strlen(command_str) == 0) {
    /* error handling for corrupt state */
}
// ...

```

تابع ابتدا با بررسی مسیر [proc/[pid] / مطمئن می شود که محفظه از قبل در حال اجرا نیست. سپس وجود دایرکتوری وضعیت را بررسی می کند. پس از این اعتبارسنجی ها، با استفاده مکرر از تابع کمکی `read_file_string`، تمام اطلاعات پیکربندی اولیه، از جمله نام ایمیج، شناسه ای لایه *OverlayFS*، دستور اجرایی، و تمام فلگ های استفاده شده (مانند `--detach` یا `--pin-cpu`) را از فایل های مربوطه در دایرکتوری وضعیت قدیمی (`old_state_dir`) می خواند و در متغیرهای محلی ذخیره می کند.

بخش دوم: بازسازی فایل سیستم و تجزیه دستور در این مرحله، فایل سیستم *OverlayFS* مجدداً مانت شده و رشته دستوری که از فایل خوانده شده، برای ارسال به `execv` آماده می شود.

```

// ... (re-mount propagate_mount_dir if it exists)

char lowerdir[PATH_MAX], upperdir[PATH_MAX], workdir[PATH_MAX],
merged[PATH_MAX];
snprintf(lowerdir, sizeof(lowerdir), "%s", image_name);
snprintf(upperdir, sizeof(upperdir), "overlay_layers/%s/upper",
overlay_id);
snprintf(workdir, sizeof(workdir), "overlay_layers/%s/work",
overlay_id);
snprintf(merged, sizeof(merged), "overlay_layers/%s/merged",
overlay_id);

char mount_opts[PATH_MAX * 3];
snprintf(mount_opts, sizeof(mount_opts), "lowerdir=%s,upperdir=%s,workdir=%s",
lowerdir, upperdir, workdir);
if (mount("overlay", merged, "overlay", 0, mount_opts) != 0) {
    /* error handling */
}

char *argv_for_container[64];
char temp_command_str[1024];
strcpy(temp_command_str, command_str);

if (strncmp(temp_command_str, "/bin/sh -c ", 11) == 0) {
    argv_for_container[0] = "/bin/sh";
    argv_for_container[1] = "-c";
    argv_for_container[2] = temp_command_str + 11;
    argv_for_container[3] = NULL;
} else {
    // Fallback to simple strtok parsing
    // ...
}

```

این بخش ابتدا با استفاده از نام ایمیج و شناسه ای `overlay_id` که از فایل وضعیت خوانده، مسیرهای لایه های *OverlayFS* را بازسازی کرده و با فراخوانی `mount`، فایل سیستم را مجدداً فعال می کند. یک نکته کلیدی در این بخش، تجزیه هی هوشمندانه ای رشته دستور است. از آنجایی که دستورات پیچیده شل (مانند یک حلقه while) در فایل وضعیت به صورت یک رشته واحد ذخیره شده اند، استفاده از `strtok` به تنها بی آنها را به اشتباه به چندین آرگومان تقسیم می کند. برای حل این مشکل، کد ابتدا بررسی می کند که آیا دستور با `/bin/sh -c` شروع می شود یا خیر. اگر چنین بود، سه آرگومان `"/bin/sh"`، `"-c"` و "کل رشته باقی مانده" را به صورت دستی می سازد. در غیر این صورت، از روش ساده تر `strtok` برای دستورات ساده استفاده می کند.

بخش سوم: ایجاد پروسه جدید و انتقال وضعیت در این مرحله، یک پروسه‌ی کاملاً جدید ایجاد شده و وضعیت از *PID* قدیمی به *PID* جدید منتقل می‌شود.

```
// ... (pipe creation and container_args setup)

pid_t new_pid = clone(container_main, stack_top, clone_flags,
&args);

// ... (error handling and user namespace mapping)

close(sync_pipe[1]); // Signal child to proceed

char new_state_dir[PATH_MAX];
snprintf(new_state_dir, sizeof(new_state_dir), "%s/%ld",
MY_RUNTIME_STATE, (long)new_pid);
rename(old_state_dir, new_state_dir);

char cgroup_path[PATH_MAX];
snprintf(cgroup_path, sizeof(cgroup_path), "%s/container_%ld",
MY_RUNTIME_CGROUP, (long)new_pid);
mkdir(cgroup_path, 0755);

// ... (re-applying resource limits to the new cgroup path)

char procs_path[PATH_MAX];
snprintf(procs_path, sizeof(procs_path), "%s/cgroup.procs",
cgroup_path);
char new_pid_str[16];
snprintf(new_pid_str, sizeof(new_pid_str), "%ld",
(long)new_pid);
write_file(procs_path, new_pid_str);
```

این بخش دقیقاً مانند تابع *do_run*، یک پروسه‌ی جدید با استفاده از *clone* و با همان فلگ‌های پیکربندی شده (مانند *share_ipc*) ایجاد می‌کند و از مکانیزم *pipe* برای همگامسازی *User Namespace* برهه می‌برد. مهم‌ترین و کلیدی‌ترین عمل در این بخش، فرآخوانی (*rename*) *old_state_dir*، *new_state_dir* است. این دستور، دایرکتوری وضعیت *PID* قدیمی به نام *PID* جدید تغییر می‌دهد. این کار تضمین می‌کند که هویت محفظه اکنون با پروسه‌ی جدید آن گره خورده است. پس از آن، یک دایرکتوری *cgroup* جدید برای *PID* جدید ساخته شده و تمام محدودیت‌های منابع که از فایل‌های وضعیت خوانده شده بودند، مجدداً در فایل‌های این *cgroup* جدید نوشته می‌شوند.

بخش چهارم: مدیریت حالت اجرایی نهایی در انتهای، تابع بر اساس وضعیت اصلی محفظه، تصمیم می‌گیرد که آیا به صورت متصل یا جدا شده اجرا شود.

```
if (original_detach_flag) {
    printf("Container %s started with new PID %ld\n", pid_str,
    (long)new_pid);
    return 0;
} else {
    printf("Container %s started with new PID %ld. Press Ctrl+C
to stop.\n",
    pid_str, (long)new_pid);
    waitpid(new_pid, NULL, 0);
    printf("Container %ld has exited. Use 'rm' to clean up.\n",
    (long)new_pid);
}
return 0;
```

این بخش با بررسی متغیر *original_detach_flag* که در ابتدای تابع از فایل وضعیت خوانده شده، رفتار نهایی را تعیین می‌کند. اگر محفظه در اجرای اولیه‌ی خود به صورت جدا شده (*detached*) اجرا شده بود، تابع *do_start* نیز پس از چاپ *PID* جدید، بلافاصله خارج می‌شود. اما اگر محفظه در حالت متصل (*attached*) بوده، تابع با استفاده از *waitpid* منتظر می‌ماند تا پروسه‌ی جدید خاتمه یابد و به این ترتیب، ترمینال کاربر مجدداً به محفظه متصل می‌شود.

۹.۲.۳ تابع *do_rm*

این تابع، که مسئولیت اجرای دستور *rm* را بر عهده دارد، آخرین مرحله در چرخه‌ی حیات یک محفظه است. وظیفه‌ی این تابع، حذف کامل و غیرقابل بازگشت تمام منابع مرتبط با یک محفوظه می‌توقف شده است. این عمل شامل پاکسازی نقاط مانت، حذف لایه‌ی قابل نوشتن (*OverlayFS*)،

حذف دایرکتوری وضعیت، و حذف دایرکتوری *cgroup* می‌شود.

```
int do_rm(int argc, char *argv[]) {
    if (argc < 2) { /* error handling */ }
    char *pid_str = argv[1];

    // Safety check: Do not remove a running container.
    char proc_path[PATH_MAX];
    snprintf(proc_path, sizeof(proc_path), "/proc/%s", pid_str);
    if (access(proc_path, F_OK) == 0) {
        fprintf(stderr, "Error: Cannot remove a running container.
        Use 'stop' first.\n");
        return 1;
    }

    printf("Removing container %s...\n", pid_str);

    // 1. Unmount filesystems.
    cleanup_mounts(atoi(pid_str));

    // 2. Remove the container's writable layer.
    char command[PATH_MAX];
    char state_dir[PATH_MAX];
    snprintf(state_dir, sizeof(state_dir), "%s/%s",
    MY_RUNTIME_STATE, pid_str);
    char overlay_id_path[PATH_MAX];
    snprintf(overlay_id_path, sizeof(overlay_id_path), "%s/overlay_id", state_dir);

    int random_id = -1;
    FILE* id_file = fopen(overlay_id_path, "r");
    if (id_file) {
        fscanf(id_file, "%d", &random_id);
        fclose(id_file);
    }
    if (random_id != -1) {
        snprintf(command, sizeof(command), "rm -rf
        overlay_layers/%d", random_id);
        system(command);
    }

    // 3. Remove the container's state directory.
    snprintf(command, sizeof(command), "rm -rf %s", state_dir);
    system(command);

    // 4. Remove the container's cgroup directory.
    char cgroup_dir[PATH_MAX];
    snprintf(cgroup_dir, sizeof(cgroup_dir), "%s/container_%s",
    MY_RUNTIME_CGROUP, pid_str);
    if (rmdir(cgroup_dir) != 0) {
        if (errno != ENOENT) {
            perror("Failed to remove cgroup directory");
        }
    }

    printf("Container %s removed.\n", pid_str);
    return 0;
}
```

روند اجرای این تابع با یک بررسی امنیتی مهم آغاز می‌شود. با استفاده از *access*. تابع چک می‌کند که آیا پروسه‌ای با *PID* داده شده در حال اجرا است یا خیر. اگر پروسه در حال اجرا باشد، تابع با یک پیغام خطا خارج می‌شود تا از حذف ناخواسته‌ی یک محفظه‌ی فعال جلوگیری کند. این کار کاربر را ملزم می‌کند تا ابتدا محفظه را با دستور *stop* متوقف کند.

پس از این بررسی، فرآیند پاکسازی در چند مرحله‌ی مجزا و با ترتیب مشخص انجام می‌شود. ابتدا، تابع *cleanup_mounts* فراخوانی می‌شود تا تمام نقاط مانت مرتبط با محفظه به درستی (*unmount*) شوند. این مرحله برای موفقیت‌آمیز بودن مراحل بعدی ضروری است. سپس، با خوشنود شناسه‌ی لایه‌ی *OverlayFS* از دایرکتوری وضعیت، دایرکتوری کامل لایه‌ی قابل نوشتن محفظه در مسیر *overlay_layers/* با

دستور سیستمی `rm -rf /run/my_runtime` نیز به همین روش حذف می‌گردد. در نهایت، دایرکتوری خالی `cgroup` محفظه با فراخوانی `rmdir` پاکسازی می‌شود. این ترتیب عملیات تضمین می‌کند که تمام ردپاهای محفوظه از روی سیستم به طور کامل و تمیز حذف شوند.

۱۰.۲.۳ main تابع

این تابع، نقطه‌ی ورود اصلی (*entry point*) برای کل برنامه است. وظیفه‌ی آن بسیار ساده اما حیاتی است: تجزیه‌ی اولین آرگومان خط فرمان برای تشخیص اینکه کاربر قصد اجرای کدام دستور را دارد و سپس هدایت اجرای برنامه به تابع مربوطه. این تابع به عنوان یک توزیع‌کننده‌ی مرکزی (*central dispatcher*) عمل می‌کند.

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <command> [args...]\n"
                "Commands: run, list, status, freeze, thaw, stop, start, rm\n",
                argv[0]);
        return 1;
    }
    if (strcmp(argv[1], "run") == 0) {
        return do_run(argc - 1, &argv[1]);
    } else if (strcmp(argv[1], "list") == 0) {
        return do_list(argc - 1, &argv[1]);
    } else if (strcmp(argv[1], "status") == 0) {
        return do_status(argc - 1, &argv[1]);
    } else if (strcmp(argv[1], "freeze") == 0) {
        return do_freeze(argc - 1, &argv[1]);
    } else if (strcmp(argv[1], "thaw") == 0) {
        return do_thaw(argc - 1, &argv[1]);
    } else if (strcmp(argv[1], "stop") == 0) {
        return do_stop(argc - 1, &argv[1]);
    } else if (strcmp(argv[1], "start") == 0) {
        return do_start(argc - 1, &argv[1]);
    } else if (strcmp(argv[1], "rm") == 0) {
        return do_rm(argc - 1, &argv[1]);
    } else {
        fprintf(stderr, "Unknown command: %s\n", argv[1]);
        return 1;
    }
    return 0;
}
```

رونده‌ی اجرای این تابع به این صورت است که ابتدا تعداد آرگومان‌های ورودی (`argc`) را بررسی می‌کند. اگر تعداد آرگومان‌ها کمتر از دو باشد (یعنی کاربر هیچ دستوری را مشخص نکرده باشد)، یک پیغام راهنمایی که لیست تمام دستورات معتبر را نشان می‌دهد، در خروجی خطای استاندارد (`stderr`) چاپ شده و برنامه با کد خطای ۱ خارج می‌شود.

در غیر این صورت، تابع با یک زنجیره‌ی `if-else`، اولین آرگومان (`argv[1]`) را با نام دستورات معتبر مقایسه می‌کند. برای این مقایسه از تابع `strcmp` استفاده می‌شود. به محض یافتن تطابق، تابع مربوط به آن دستور (مثلاً `do_run` برای دستور `run`) فراخوانی می‌شود. یک نکته‌ی مهم در اینجا، نحوه‌ی ارسال آرگومان‌ها به توابع `*do_*` است. با ارسال `argc - 1` و `&argv[1]`، ما به طور موثر آرایه‌ی آرگومان‌ها را "شیفت" می‌دهیم، به طوری که از دید تابع فراخوانی شده، نام دستور (مانند `run`) دیگر وجود ندارد و اولین آرگومان، فلگ‌ها یا پارامترهای بعدی خواهد بود. این کار تجزیه‌ی آرگومان‌ها را در هر تابع ساده‌تر می‌کند. اگر هیچ‌کدام از دستورات مطابقت نداشته باشند، یک پیغام خطای "دستور ناشناخته" چاپ شده و برنامه خاتمه می‌یابد.

۱۱.۲.۳ setup_rootfs.sh

این اسکریپت بیش مسئولیت ایجاد یک فایل سیستم ریشه‌ی حداقلی و مستقل را بر عهده دارد که به عنوان "ایمیج پایه" (`ubuntu-base-image`) برای تمام محفظه‌ها عمل می‌کند. اجرای موقتی آمیز این اسکریپت، یک پیش‌نیاز برای اجرای برنامه است. این اسکریپت به چند بخش منطقی تقسیم می‌شود.

بخش اول: آماده‌سازی سیستم و پیش‌نیازها در ابتدای اسکریپت، چند دستور برای آماده‌سازی محیط سیستم میزبان اجرا می‌شود. این دستورات اطمینان حاصل می‌کنند که زیرساخت‌های لازم برای اجرای برنامه و مدیریت `cgroup`ها فراهم است.

```
#!/bin/bash
set -e

# --- Part 1: Cgroup and Runtime State Setup ---
echo "--> Ensuring runtime directories exist..."
```

```

mkdir -p /sys/fs/cgroup/my_runtime
mkdir -p /run/my_runtime

echo "--> Enabling CPU, IO, and Memory cgroup controllers..."
echo "+cpu +io +memory +pids" | sudo tee
/sys/fs/cgroup/cgroup.subtree_control > /dev/null 2>&1 || true

```

دستور `e - set` تضمین می‌کند که اسکریپت در صورت بروز هرگونه خطا بلافضلله متوقف شود. سپس، با استفاده از `-p`، دایرکتوری‌های اصلی برنامه در مسیر ریشه‌ی `cgroups` و همچنین دایرکتوری وضعیت در مسیر `/run` / ایجاد می‌شوند. مهم‌ترین بخش در اینجا، اجرای دستور `cgroup.subtree_control` است. این دستور، کنترل‌های منابع مورد نیاز را در فایل `cgroupsv2` یک پیش‌نیاز ضروری است و به برنامه‌ی ما اجازه می‌دهد تا این کنترل‌ها را برای زیرگروه‌هایی که برای هر محفظه ایجاد می‌کند، به ارت برده و مدیریت نماید.

بخش دوم: ایجاد ساختار ایمیج پایه این بخش، ساختار اولیه و فایل‌های پیکربندی اصلی را برای ایمیج پایه ایجاد می‌کند.

```

IMAGE_DIR="ubuntu-base-image"

if [ -d "$IMAGE_DIR" ]; then
    echo "--> Base image '$IMAGE_DIR' already exists. Skipping..." 
    exit 0
fi

echo "--> Creating base image at ./$IMAGE_DIR"

COMMANDS=()
/bin/bash "/bin/ls" "/bin/cat" "/bin/echo" "/bin/ps" #... and
others
)

mkdir -p "${IMAGE_DIR}"/{bin,lib,lib64,usr/bin,proc,tmp,dev,etc,root}

echo "--> Creating /etc/passwd and /etc/group"
echo "root:x:0:0:root:/bin/bash" > "${IMAGE_DIR}/etc/passwd"
echo "root:x:0:" > "${IMAGE_DIR}/etc/group"
echo "nogroup:x:65534:" >> "${IMAGE_DIR}/etc/group"

```

اسکریپت ابتدا با یک شرط `if` بررسی می‌کند که آیا دایرکتوری `ubuntu-base-image` از قبل وجود دارد یا خیر. این کار از اجرای مجدد و بازنویسی ناخواسته‌ی ایمیج جلوگیری می‌کند. سپس، آرایه‌ی `COMMANDS` لیستی از تمام ابزارهای خط فرمانی که باید در داخل محفظه در دسترس باشند را تعریف می‌کند. پس از آن، ساختار دایرکتوری استاندارد لینوکس با دستور `mkdir` ایجاد می‌شود. در نهایت، فایل‌های ضروری `/etc/passwd` و `/etc/group` با محتوای حداقلی ساخته می‌شوند. این فایل‌ها برای عملکرد صحیح `User Namespace` و ابزارهایی مانند `whoami` نگاشت شناسه‌ی کاربری `UID` به نام کاربری دارند، حیاتی هستند.

بخش سوم: کامپایل درجا و کپی وابستگی‌ها این بخش شامل تابع کلیدی `copy_binary_with_deps` است که وظیفه‌ی اصلی پر کردن ایمیج با ابزارهای قابل اجرا را بر عهده دارد.

```

copy_binary_with_deps() {
    local binary_path="$1"
    local source_path="$binary_path"
    # ... (logic to handle local vs system binaries)
    if [ ! -f "$source_path" ]; then
        echo "--> WARNING: Command not found: $source_path"
        return
    fi
    # ... (mkdir and cp the binary itself)
    cp "$source_path" "$dest_binary"

    # Copy dependencies
    for lib in $(ldd "$source_path" | awk 'NF == 4 {print $3}; NF
== 2 {print $1}' | grep -v "not a dynamic executable"); do
        if [ ! -f "$lib" ]; then continue; fi
        local dest_lib="${IMAGE_DIR}${lib}"
        if [ ! -f "$dest_lib" ]; then
            mkdir -p "$(dirname "$dest_lib")"
            cp "$lib" "$dest_lib"
        fi
    done
}

```

```

        fi
    done
}

for cmd in "${COMMANDS[@]}"; do
    copy_binary_with_deps "$cmd"
done

```

این تابع در یک حلقه برای تمام دستورات تعریف شده در آرایه `COMMANDS` فراخوانی می شود. برای هر دستور، ابتدا خود فایل باینری را در مسیر مربوطه در داخل ایمیج کپی می کند. سپس، با استفاده از دستور `ldd`، لیستی از تمام کتابخانه های اشتراکی (*shared libraries*) که آن باینری برای اجرا به آن ها نیاز دارد را استخراج می کند. خروجی `ldd` با ابزارهایی مانند `awk` و `grep` پردازش می شود تا فقط مسیر کتابخانه ها به دست آید. در نهایت، هر یک از این کتابخانه های وابسته نیز در مسیر صحیح خود در داخل ایمیج کپی می شوند. این فرآیند تضمین می کند که تمام ابزارها در محیط ایزوله `chroot` که به کتابخانه های سیستم میزبان دسترسی ندارد، به درستی کار کنند.

بخش چهارم: تنظیمات نهایی ایمیج در انتهای اسکریپت، چند دستور نهایی برای تکمیل ایمیج اجرا می شود.

```

echo "--> Creating /bin/sh symlink to /bin/bash...
ln -sf /bin/bash "${IMAGE_DIR}/bin/sh"

echo "--> Creating device nodes in ${IMAGE_DIR}/dev...
mknod -m 666 "${IMAGE_DIR}/dev/zero" c 1 5
mknod -m 660 "${IMAGE_DIR}/dev/sda" b 8 0

```

ابتدا، یک لینک نمادین (*symlink*) از `/bin/sh` به `/bin/bash` ایجاد می شود. این کار برای سازگاری با بسیاری از اسکریپت های شل که انتظار دارند `sh` را باشد، ضروری است. سپس، با استفاده از دستور `mknod`، فایل های دستگاه (*device nodes*) حداقلی ایجاد می شوند. `/dev/zero` یک دستگاه کاراکتری (*character device*) است که جریانی بی پایان از بایت های صفر تولید می کند و برای تست های ورودی/خروجی استفاده می شود. `/dev/sda` نیز به عنوان یک دستگاه بلاکی *block device* شبیه سازی شده برای تست محدودیت های *I/O* ایجاد می شود.

۱۲.۲.۳ اسکریپت ناظارتی `monitor.py`

این اسکریپت پایتون به عنوان یک ابزار ناظارتی مستقل عمل کرده و با استفاده از فناوری *eBPF*، فراخوانی های سیستمی مرتبط با ایجاد محفظه را در سطح هسته لینوکس رهگیری و ثبت می کند. این اسکریپت از کتابخانه `bcc` برای ساده سازی فرآیند کامپایل و بارگذاری برنامه های *bpf* بهره می برد.

بخش اول: کد `eBPF` به زبان C هسته ای اصلی منطق ناظارتی، یک برنامه کوچک به زبان C است که به صورت یک رشته ای چند خطی در داخل اسکریپت پایتون تعریف شده است. این کد در زمان اجرا توسط `bcc` کامپایل شده و در فضای هسته *kernel-space* بارگذاری می شود.

```

//bpf_text = """
#include <linux/sched.h>
#include <uapi/linux/bpf.h>

struct data_t {
    u32 pid;
    char comm[TASK_COMM_LEN];
    char syscall_name[32];
};

BPF_PERF_OUTPUT(events);

#define COPY_SYSCALL_NAME(name)
__builtin_memcpy(&data.syscall_name, name, sizeof(name))

// --- PROBE 1: For the 'clone' syscall ---
TRACEPOINT_PROBE(syscalls, sys_enter_clone) {
    char comm[TASK_COMM_LEN];
    bpf_get_current_comm(&comm, sizeof(comm));

    char target_comm[] = "my_runner";
    for (int i = 0; i < sizeof(target_comm) - 1; ++i) {
        if (comm[i] != target_comm[i]) {
            return 0;
        }
    }

    unsigned long clone_flags = args->clone_flags;
}
```

```

if (clone_flags & (CLONE_NEWNS | CLONE_NEWCGROUP | CLONE_NEWUTS
| CLONE_NEWIPC | CLONE_NEWUSER | CLONE_NEWPID | CLONE_NEWWNET)) {
    struct data_t data = {};
    data.pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&data.comm, sizeof(data.comm));

    char name[] = "clone (new ns)";
    COPY_SYSCALL_NAME(name);

    events.perf_submit(args, &data, sizeof(data));
}
return 0;
}

// --- PROBE 2: For the 'mkdir' syscall ---
TRACEPOINT_PROBE(syscalls, sys_enter_mkdir) {
    char comm[TASK_COMM_LEN];
    bpf_get_current_comm(&comm, sizeof(comm));

    char target_comm[] = "my_runner";
    for (int i = 0; i < sizeof(target_comm) - 1; ++i) {
        if (comm[i] != target_comm[i]) {
            return 0;
        }
    }

    const char* path = (const char*)args->pathname;
    char cgroup_path[] = "/sys/fs/cgroup";
    char path_buf[15];
    bpf_probe_read_user_str(&path_buf, sizeof(path_buf), path);

    for (int i = 0; i < sizeof(cgroup_path) - 1; ++i) {
        if (path_buf[i] != cgroup_path[i]) {
            return 0;
        }
    }

    struct data_t data = {};
    data.pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&data.comm, sizeof(data.comm));

    char name[] = "mkdir (cgroup)";
    COPY_SYSCALL_NAME(name);

    events.perf_submit(args, &data, sizeof(data));
    return 0;
}
"""


```

این کد ابتدا ساختار داده‌ی `data_t` را تعریف می‌کند که برای ارسال اطلاعات مربوط به یک رویداد (شامل `PID` پروسه، نام پروسه و نام فراخوانی سیستمی) از هسته به فضای کاربری استفاده می‌شود. (BPF_PERF_OUTPUT(events) یک کانال ارتباطی به نام `events` است و برای ارسال کارآمد داده‌ها به کار می‌رود. سپس، دو "پروب" (probe) با استفاده از مکروی `TRACEPOINT_PROBE` تعریف می‌شوند. این پروب‌ها به نقاط رهگیری (tracepoints) پایدار و از پیش تعریف شده در هسته متصل می‌شوند: یکی به نقطه‌ی شروع فراخوانی سیستمی `clone` و دیگری به `mkdir` در داخل هر پروب، ابتدا با استفاده از `bpf_get_current_comm` نام پروسه‌ی جاری دریافت شده و با نام "my_runner" مقایسه می‌شود.

این یک فیلتر سیار مهم است که از ثبت رویدادهای مربوط به سایر پروسه‌های سیستم جلوگیری کرده و تنها فعالیت‌های برنامه‌ی ما را ثبت می‌کند. در پروب `clone`، پرچم‌های ورودی بررسی می‌شوند تا اطمینان حاصل شود که این فراخوانی با هدف ایجاد یک `namespace` جدید انجام شده است. در پروب `mkdir`، مسیر ورودی خوانده شده و بررسی می‌شود که آیا با `/sys/fs/cgroup` شروع می‌شود یا خیر. در صورت برقراری شرایط، ساختار `data_t` با اطلاعات مربوطه پر شده و با فراخوانی `events.perf_submit` به فضای کاربری ارسال می‌شود.

بخش دوم: ساختار داده و تابع `Callback` در پایتون برای دریافت و پردازش داده‌های ارسالی از هسته، یک بخش معادل در پایتون تعریف می‌شود.

```
class Data(ct.Structure):
```

```

_fields_ = [
    ("pid", ct.c_uint), ("comm", ct.c_char * 16),
    ("syscall_name", ct.c_char * 32),
]

def print_event(cpu, data, size):
    event = ct.cast(data, ct.POINTER(Data)).contents
    current_time = time.strftime("%Y-%m-%d %H:%M:%S")
    comm = event.comm.decode('utf-8', 'replace')
    syscall = event.syscall_name.decode('utf-8', 'replace')
    log_line = f"{current_time} | PID: {event.pid:<7} | COMM: {comm:<15} | SYSCALL: {syscall}\n"
    print(log_line, end="")
    with open("ebpf_log.txt", "a") as f:
        f.write(log_line)

```

کلاس Data که از `ctypes.Structure` ارث بری می‌کند، یک ساختار داده در پایتون تعریف می‌کند که طرح بندی آن دقیقاً با ساختار `t` در کد C مطابقت دارد. این تطابق به کتابخانه `ctypes` اجازه می‌دهد تا بایت‌های خامی که از هسته دریافت می‌شوند را به درستی به یک شیء پایتون قابل استفاده تبدیل کند.

تابع `print_event` به عنوان یک تابع بازخوانی (`callback`) عمل می‌کند. این تابع هر بار که یک رویداد جدید از هسته به بافر می‌رسد، به طور خودکار توسط کتابخانه `bcc` فراخوانی می‌شود. در داخل این تابع، داده‌های خام به شیء یا `Data` تبدیل شده، رشته‌های بایتی به رشته‌های متغیر استاندارد پایتون (utf-8) تبدیل می‌شوند، و در نهایت یک خط لاتگ با فرمت‌بندی مناسب (شامل تاریخ و زمان) ساخته شده و هم در کنسول چاپ و هم در فایل `ebpf_log.txt` نوشته می‌شود.

بخش سوم: بلوک اصلی اجرای اسکریپت این بخش نهایی، مسئولیت کامپایل، بارگذاری و مدیریت حلقه‌ی رویداد برنامه‌ی eBPF را بر عهده دارد.

```

print("Starting eBPF monitoring... Press Ctrl+C to exit.")
try:
    cflags = ["-Wno-macro-redefined"]
    b = BPF(text=bpf_text, cflags=cflags)

    b["events"].open_perf_buffer(print_event)
    while True:
        try:
            b.perf_buffer_poll()
        except KeyboardInterrupt:
            exit()

except Exception as e:
    print(f"Error: {e}")

```

در این بلوک، ابتدا شیء اصلی BPF از کتابخانه `bcc` با ارسال کد C به عنوان متن، ایجاد می‌شود. پارامتر `cflags` برای ارسال فلگ‌های اضافی به کامپایلر `Clang/LLVM` استفاده می‌شود تا از نمایش هشدارهای بی‌خطر مربوط به تعریف ماکروها جلوگیری شود. سپس، با استفاده از (که در کد C تعریف شده بود) به تابع بازخوانی `print_event` در پایتون متصل می‌شود.

در نهایت، اسکریپت وارد یک حلقه‌ی بی‌نهایت می‌شود و به طور مداوم تابع `(b.perf_buffer_poll()` را فراخوانی می‌کند. این تابع منتظر رسیدن داده‌های جدید از هسته می‌ماند و به مخصوص دریافت، تابع بازخوانی مربوطه را اجرا می‌کند. این حلقه تا زمانی که کاربر با فشردن `Ctrl+C` برنامه را متوقف کند، ادامه می‌یابد.

۳.۳ نتایج آزمایش‌ها

در این بخش، مجموعه‌ای از آزمایش‌های عملی برای اعتبارسنجی و نمایش صحت عملکرد قابلیت‌های کلیدی سیستم مدیریت محفظه ارائه می‌شود. هر آزمایش به صورت یک گزارش مجزا طراحی شده است که شامل هدف، مراحل اجرا، و نتایج مورد انتظار است. این آزمایش‌ها به صورت گام به گام، از مفاهیم پایه‌ای مانند معماری سیستم و ایزوله‌سازی گرفته تا قابلیت‌های پیشرفته مدیریتی و بهینه‌سازی عملکرد را پوشش می‌دهند. هدف نهایی، ارائه‌ی شواهد مستند برای اثبات این است که سیستم پیاده‌سازی شده، یک ابزار کارا، پایدار و امن برای مدیریت چرخه‌ی حیات محفظه‌ها است.

توجه: فایل `main.c` باید تحت عنوان `my_runner` کامپایل شود.

```
sudo -w -o my_runner main.c
```

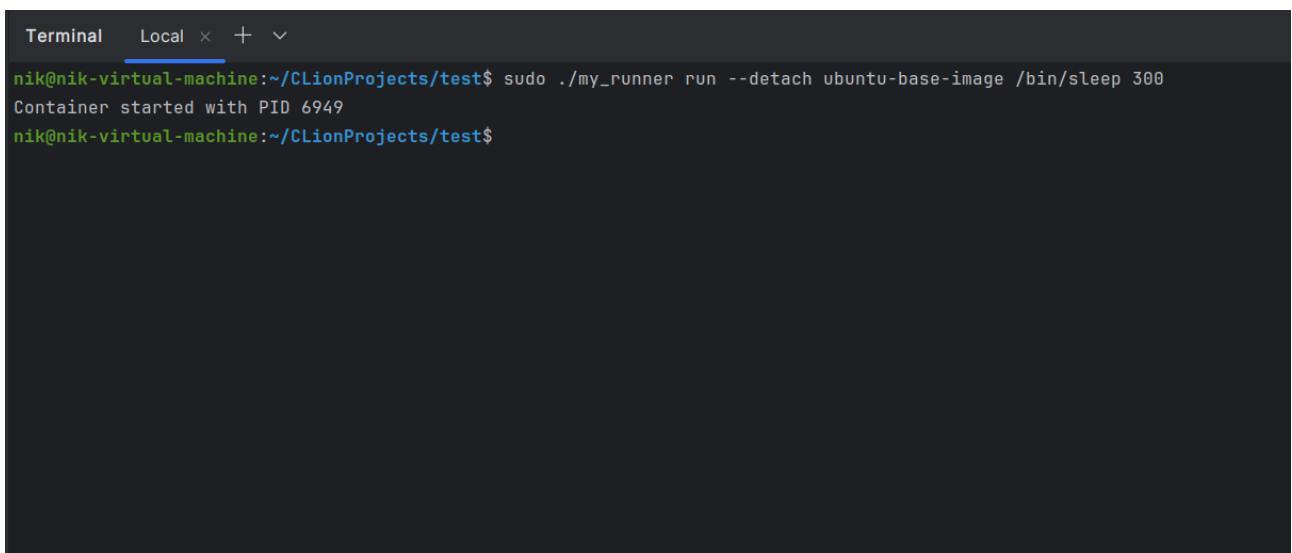
۱.۳.۳ آزمایش ۱: تأیید معماری بدون دیمون (Daemonless)

هدف هدف این تست، تأیید و نمایش معماری *daemonless* سیستم مدیریت محفظه است. در این معماری، هیچ پروسه‌ی پس‌زمینه‌ی دائمی برای مدیریت محفظه‌ها وجود ندارد و هر دستور به صورت یک پروسه‌ی مستقل و موقتی اجرا می‌شود. این آزمایش نشان می‌دهد که پس از اجرای یک محفظه در حالت جدا شده *detached*، پروسه‌ی اصلی برنامه (`my_runner`) خاتمه یافته و محفظه به صورت مستقل به کار خود ادامه می‌دهد. این طراحی با حذف نقطه‌ی شکست واحد، پایداری سیستم را افزایش می‌دهد.

مراحل اجرا و تأیید صحت مرحله اول: اجرای یک محفظه در حالت **Detached** یک محفظه با استفاده از دستور زیر اجرا می‌کنیم. این محفظه صرفاً دستور `sleep` را برای مدت ۳۰۰ ثانیه اجرا می‌کند. فلگ `--detach` باعث می‌شود که `my_runner` پس از ایجاد محفظه، بلافصله خارج شود.

```
sudo ./my_runner run --detach ubuntu-base-image /bin/sleep 300
```

نتیجه: دستور بلافصله پس از اجرا، *PID* محفظه را چاپ کرده و به پایان می‌رسد.



The screenshot shows a terminal window with the title 'Terminal'. The command entered is 'sudo ./my_runner run --detach ubuntu-base-image /bin/sleep 300'. The output shows the container starting with PID 6949. The terminal then remains blank, indicating the process has detached and is running in the background.

شکل ۷: اجرای محفظه در حالت *detached* و دریافت *PID*.

مرحله دوم: بررسی پروسه‌های در حال اجرای هاست

اکنون بررسی می‌کنیم که آیا پروسه‌ی مدیریتی دائمی در سیستم وجود دارد یا خیر. ابتدا به دنبال هرگونه پروسه‌ی `my_runner` در حال اجرا می‌گردیم.

```
ps -ef | grep my_runner
```

نتیجه: هیچ پروسه‌ای با نام `my_runner` یافت نمی‌شود (به جز خود دستور `grep`). این نتیجه ثابت می‌کند که ابزار ما پس از انجام وظیفه‌اش، از سیستم خارج شده و به صورت یک دیمون در پس‌زمینه باقی نمانده است. (تصاویر در صفحه بعد آورده شده‌اند.) سپس، پروسه‌ی اصلی محفظه را جستجو می‌کنیم.

```
ps -ef | grep "sleep 300"
```

نتیجه: پروسه‌ی `sleep 300` با موفقیت پیدا می‌شود. (PPID) این پروسه، عدد ۱ (پروسه‌ی `init` سیستم) یا پروسه‌ی دیگری از سیستم خواهد بود، که نشان می‌دهد محفظه به صورت مستقل در حال اجراست.

نتیجه‌گیری آزمایش نتایج به وضوح نشان می‌دهند که پروسه‌ی `my_runner` پس از اجرای محفظه خاتمه یافته و پروسه‌ی محفظه به صورت مستقل و به عنوان فرزند یکی از پروسه‌های اصلی سیستم به کار خود ادامه می‌دهد. این مشاهدات، معماری بدون دیمون سیستم را تأیید می‌کنند.

```

Terminal Local + ▾
nik@nik-virtual-machine:~/CLionProjects/test$ ps -ef | grep my_runner
nik      7061  6922  0 19:53 pts/5    00:00:00 grep --color=auto my_runner
nik@nik-virtual-machine:~/CLionProjects/test$


Terminal Local + ▾
nik@nik-virtual-machine:~/CLionProjects/test$ ps -ef | grep my_runner
nik      7061  6922  0 19:53 pts/5    00:00:00 grep --color=auto my_runner
nik@nik-virtual-machine:~/CLionProjects/test$ ps -ef | grep "sleep 300"
root     6949  1033  0 19:52 ?        00:00:00 /bin/sleep 300
nik      7083  6922  0 19:54 pts/5    00:00:00 grep --color=auto sleep 300
nik@nik-virtual-machine:~/CLionProjects/test$
```

شکل ۸: تأیید عدم وجود پروسه‌ی my_runner و مستقل بودن پروسه‌ی محفظه.

۲.۳.۳ آزمایش ۲: ایزوله‌سازی فایل‌سیستم با chroot

هدف هدف این تست، تأیید این است که سیستم مدیریت محفظه به درستی از فراخوانی سیستمی chroot برای ایجاد یک "زندان" فایل‌سیستم (filesystem jail) استفاده می‌کند. این قابلیت امنیتی بنیادی، دید یک محفظه به فایل‌سیستم را به دایرکتوری ریشه‌ی خودش (rootfs) محدود می‌کند. این آزمایش نشان می‌دهد که یک پروسه‌ی در حال اجرا درون محفظه، قادر به مشاهده یا دسترسی به فایل‌ها و دایرکتوری‌های سیستم میزبان نیست.

مراحل اجرا و تأیید صحت مرحله اول: ایجاد یک فایل محترمانه روی هاست
ابتدا، یک فایل متنی در یک مسیر استاندارد روی سیستم میزبان ایجاد می‌کنیم. این فایل نباید از داخل محفظه قابل دسترس باشد.

```
echo "This is a secret file on the host" > /tmp/host_secret_file.txt
```

```

Terminal Local + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ echo "This is a secret file on the host" > /tmp/host_secret_file.txt
nik@nik-virtual-machine:~/CLionProjects/os$ cat /tmp/host_secret_file.txt
This is a secret file on the host
nik@nik-virtual-machine:~/CLionProjects/os$
```

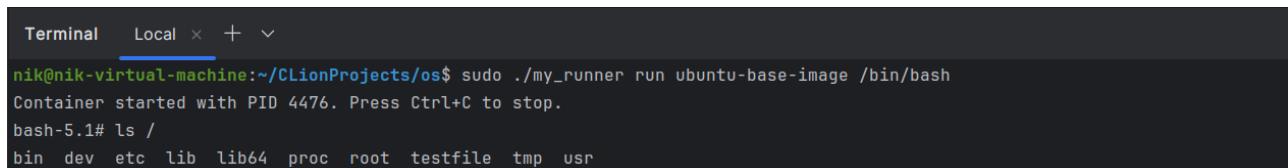
شکل ۹: ایجاد یک فایل روی هاست

مرحله دوم: اجرای محفظه و بررسی ریشه فایل سیستم
یک محفظه با شل تعاملی bash اجرا می‌کنیم. سپس در داخل محفظه، محتويات دایرکتوری ریشه (/) را لیست می‌کنیم.

```
# Command to start the container
sudo ./my_runner run ubuntu-base-image /bin/bash
```

```
# Command inside the container
ls /
```

نتیجه: خروجی دستور ls، مجموعه‌ی حداقلی از دایرکتوری‌هایی است که توسط اسکریپت setup_rootfs.sh ایجاد شده‌اند (مانند bin، dev، etc). این خروجی با محتويات دایرکتوری ریشه‌ی هاست کاملاً متفاوت است.



```
Terminal Local × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run ubuntu-base-image /bin/bash
Container started with PID 4476. Press Ctrl+C to stop.
bash-5.1# ls /
bin dev etc lib lib64 proc root testfile tmp usr
```

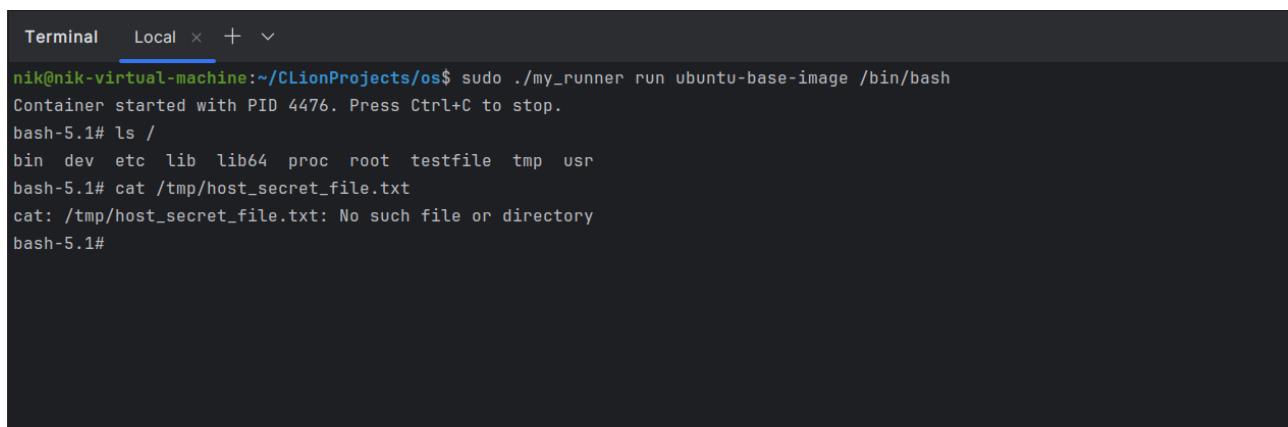
شکل ۱۰: نمایش فایل سیستم ریشه‌ی ایزوله شده در داخل محفظه.

مرحله سوم: تلاش برای دسترسی به فایل هاست

از داخل محفظه، تلاش می‌کنیم تا فایل محرمانه‌ای که روی هاست ایجاد کردیم را بخوانیم.

```
cat /tmp/host_secret_file.txt
```

نتیجه: این دستور باید با شکست مواجه شود. این شکست، اثبات قطعی عملکرد صحیح chroot است. دایرکتوری tmp / محفظه، بخشی از فایل سیستم ایزوله خودش است و هیچ ارتباطی با دایرکتوری tmp / هاست ندارد.



```
Terminal Local × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run ubuntu-base-image /bin/bash
Container started with PID 4476. Press Ctrl+C to stop.
bash-5.1# ls /
bin dev etc lib lib64 proc root testfile tmp usr
bash-5.1# cat /tmp/host_secret_file.txt
cat: /tmp/host_secret_file.txt: No such file or directory
bash-5.1#
```

شکل ۱۱: عدم موفقیت در دسترسی به فایل هاست از داخل محفظه.

نتیجه‌گیری آزمایش فراخوانی سیستمی chroot به درستی دسترسی فایل سیستم محفوظه را به دایرکتوری ریشه‌ی خودش محدود کرده است. محفوظه قادر به مشاهده یا دسترسی به فایل موجود در سیستم میزبان نبود، که این امر ایجاد یک زندان فایل سیستم امن را تأیید می‌کند.

۳.۳.۳ آزمایش ۳: سیستم فایل Union با OverlayFS

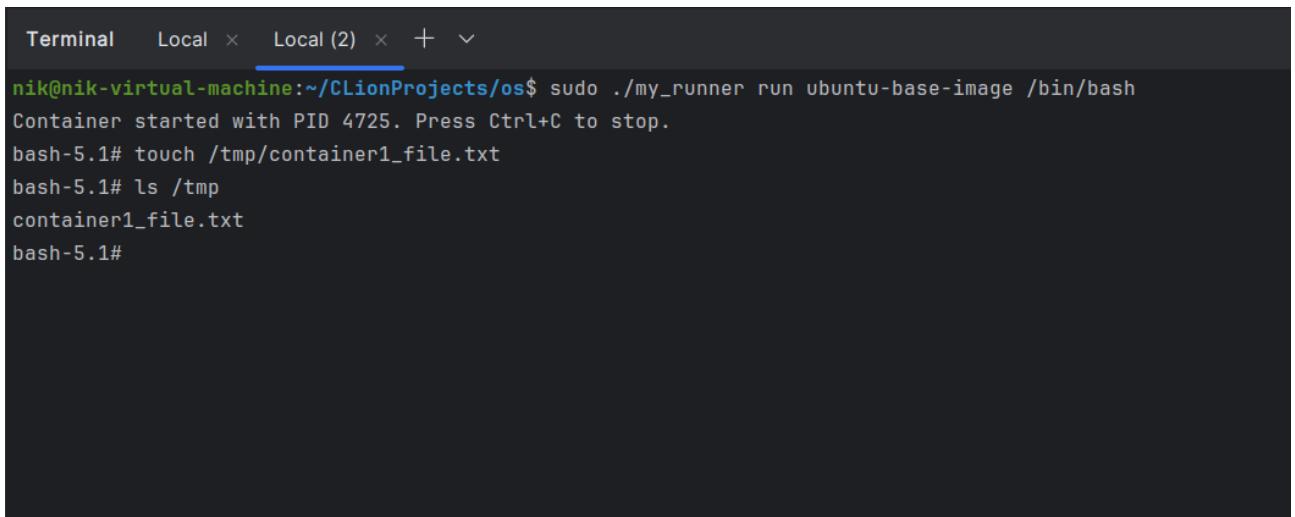
هدف هدف این تست، تأیید صحت پیاده‌سازی سیستم فایل (Union File System) با استفاده از OverlayFS است. این فناوری برای ایجاد فایل سیستم‌های ایزوله و کارآمد برای محفظه‌ها، یک جزء بنیادی محسوب می‌شود. این آزمایش نشان می‌دهد که چگونه یک لایه‌ی قابل نوشت و موقتی (upperdir) بر روی یک ایمیج پایه‌ی فقط‌خواندنی (lowerdir) قرار می‌گیرد و تغییرات هر محفظه را به صورت مجزا ذخیره می‌کند.

مراحل اجرا و تأیید صحت مرحله اول: اجرای محفظه اول و ایجاد یک فایل ابتدا، یک محفوظه با استفاده از ایمیج پایه‌ی ubuntu-base-image اجرا می‌کنیم. سپس در داخل آن، یک فایل جدید در مسیر tmp / ایجاد می‌نماییم.

```
# Command to start the container
sudo ./my_runner run ubuntu-base-image /bin/bash
```

```
# Command inside Container 1
touch /tmp/container1_file.txt
ls /tmp
```

نتیجه: دستور `ls` نشان می‌دهد که فایل `container1_file.txt` با موفقیت در داخل محفظه ایجاد شده است.



```
Terminal Local × Local (2) × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run ubuntu-base-image /bin/bash
Container started with PID 4725. Press Ctrl+C to stop.
bash-5.1# touch /tmp/container1_file.txt
bash-5.1# ls /tmp
container1_file.txt
bash-5.1#
```

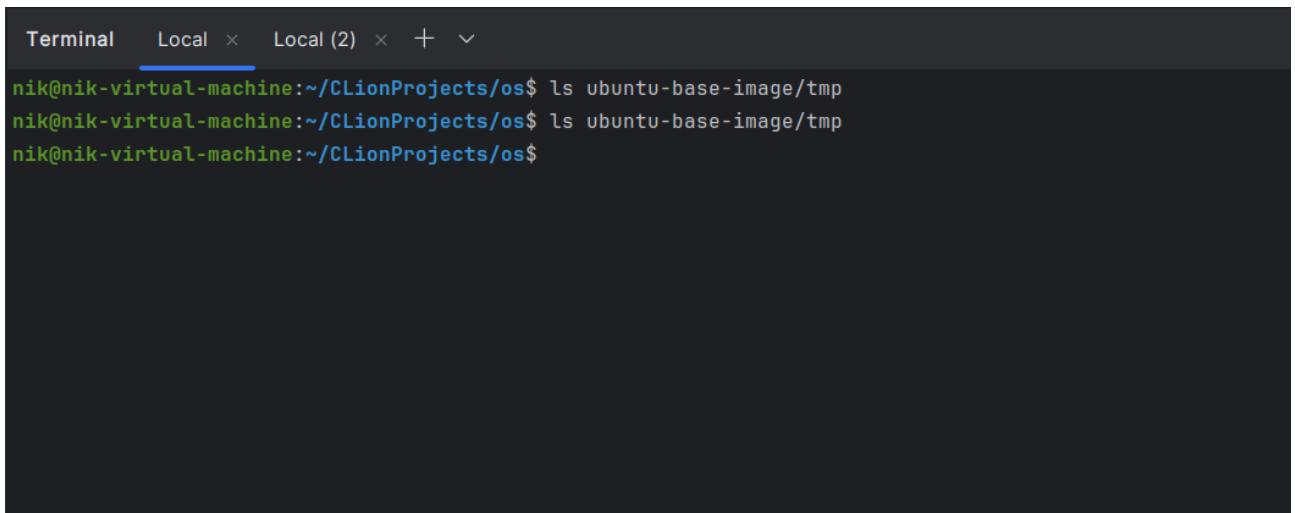
شکل ۱۲: ایجاد فایل در لایه‌ی قابل نوشتن محفظه‌ی اول.

مرحله دوم: بررسی یکپارچگی ایمیج پایه

اکنون به سیستم میزبان بازگشته و محتويات دایرکتوری `tmp` را در داخل ایمیج پایه‌ی اصلی بررسی می‌کنیم.

```
ls ubuntu-base-image/tmp
```

نتیجه: این دایرکتوری همچنان خالی است. این نتیجه، اثبات قطعی این است که عملیات نوشتن در داخل محفظه، هیچ تغییری در ایمیج پایه‌ی فقط خواندنی ایجاد نکرده است.



```
Terminal Local × Local (2) × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ ls ubuntu-base-image/tmp
nik@nik-virtual-machine:~/CLionProjects/os$ ls ubuntu-base-image/tmp
nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۱۳: فایل در هاست مشاهده نمی‌شود.

مرحله سوم: بررسی لایه‌ی قابل نوشتن محفظه

در این مرحله، محتويات لایه‌ی قابل نوشتن (`upperdir`) که به صورت اختصاصی برای محفظه‌ی اول ایجاد شده را بررسی می‌کنیم.

```
# First, find the overlay_id from the container's state directory
# Then, inspect the upperdir
ls overlay_layers/[overlay_id]/upper/tmp
```

نتیجه: فایل `container1_file.txt` در این مسیر وجود دارد. این نشان می‌دهد که تغییرات به درستی در لایه‌ی اختصاصی محفوظه ذخیره شده‌اند.

```

Terminal Local × Local (2) × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner list
CONTAINER PID STATUS COMMAND
4725 Running /bin/bash
nik@nik-virtual-machine:~/CLionProjects/os$ cat /run/my_runtime/4725/overlay_id
2436nik@nik-virtual-machine:~/CLionProjects/os$ ls overlay_layers/2436/upper/tmp
container1_file.txt
nik@nik-virtual-machine:~/CLionProjects/os$

```

شکل ۱۴: ذخیره شدن تغییرات در لایه‌ی قابل نوشتن اختصاصی محفظه.

مرحله چهارم: بررسی ایزوله‌سازی بین محفظه‌ها
یک محفظه‌ی دوم از همان ایمیج پایی ubuntu-base-image اجرا کرده و محتویات دایرکتوری tmp/ آن را بررسی می‌کنیم.

```

# Command to start the second container
sudo ./my_runner run ubuntu-base-image /bin/bash

# Command inside Container 2
ls /tmp

```

نتیجه: دایرکتوری tmp/ در محفظه‌ی دوم خالی است. فایل ایجاد شده توسط محفظه‌ی اول در اینجا قابل مشاهده نیست. این امر به وضوح ثابت می‌کند که هر محفظه لایه‌ی قابل نوشتن ایزوله‌ی خود را دارد.

```

Terminal Local × Local (2) × Local (3) × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run ubuntu-base-image /bin/bash
Container started with PID 4912. Press Ctrl+C to stop.
bash-5.1# ls /tmp
bash-5.1#

```

شکل ۱۵: عدم وجود فایل ساخته شده در فایل سیستم محفظه دوم

نتیجه‌گیری آزمایش نتایج این آزمایش، پیاده‌سازی صحیح و کارآمد OverlayFS را تأیید می‌کند. این سیستم با فراهم کردن یک لایه‌ی قابل نوشتن مجزا برای هر محفظه، ضمن حفظ یکپارچگی ایمیج پایه، ایزوله‌سازی کامل بین محفظه‌ها را تضمین می‌کند. این معماری، اساس مدیریت بهینه و مدرن فایل سیستم در سیستم‌های کانترینری است.

۴.۳.۴ آزمایش ۴: ایزولهسازی جامع با Namespace

هدف هدف این تست، ارائه یک اعتبارسنجی کامل از قابلیت‌های ایزولهسازی سیستم با بررسی تمام *namespace*‌های پیاده‌سازی شده است. این آزمایش نشان می‌دهد که چگونه هر محفظه، دیدگاه مجازی و کاملاً مستقل خود را نسبت به منابع کلیدی سیستم، از جمله پروسه‌ها، شبکه، نام میزبان، کاربران و ارتباطات بین فرآیندی به دست می‌آورد.

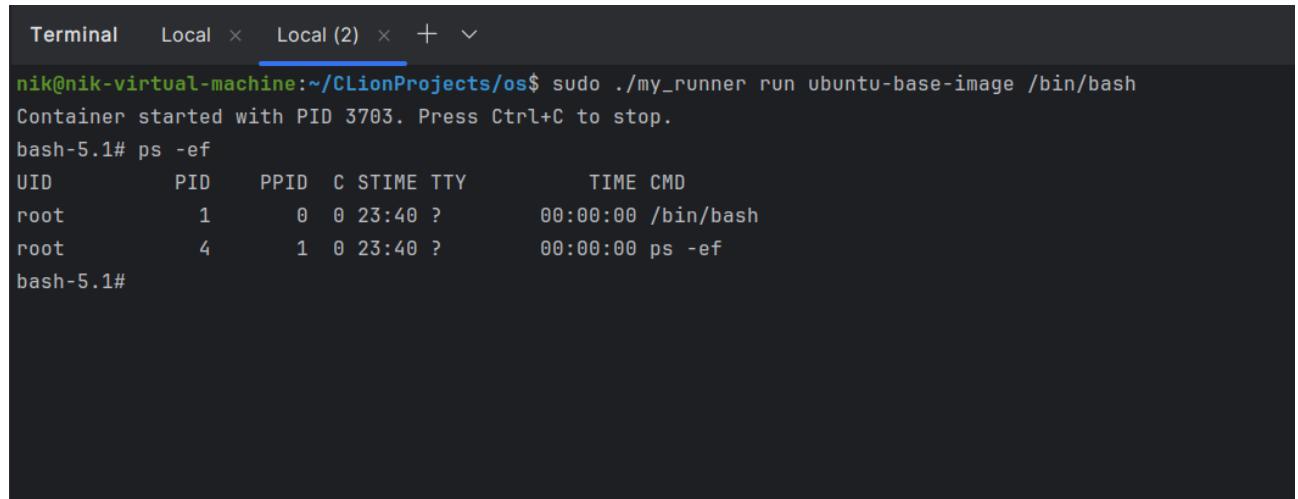
مراحل اجرا و تأیید صحت برای این تست، یک محفظه با شل تعاملی اجرا کرده و وضعیت منابع آن را با وضعیت سیستم میزبان مقایسه می‌کنیم.

مرحله اول: بررسی Namespace PID (ایزولهسازی پروسه)

ابتدا، پرسه‌ی اصلی داخل محفظه را بررسی می‌کنیم.

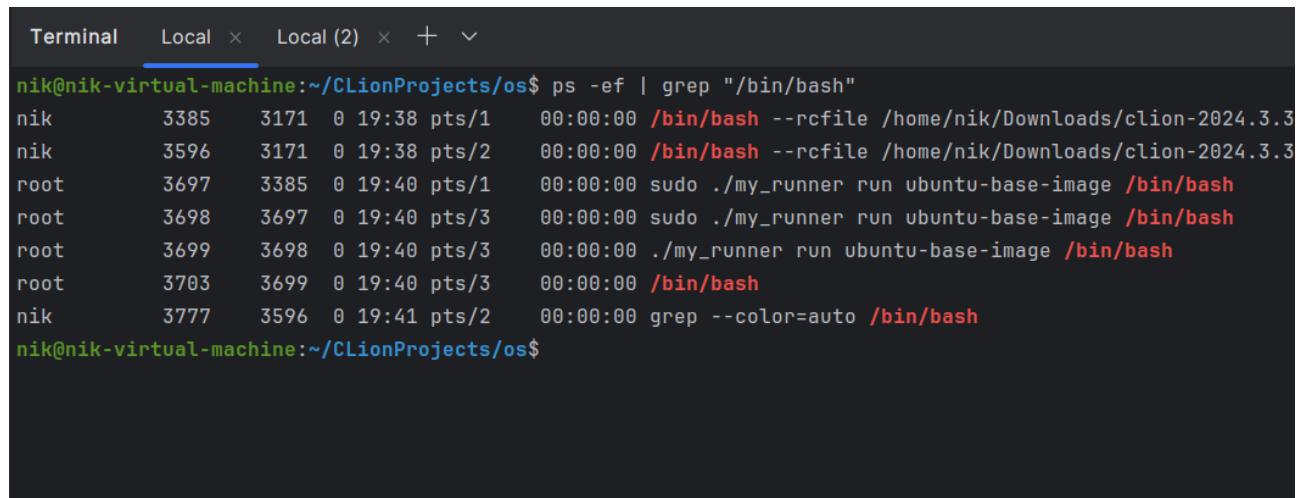
```
# Command inside the container
ps -ef
```

نتیجه: خروجی نشان می‌دهد که پرسه‌ی /bin/bash/ دارای شناسه‌ی پرسه‌ی ۱ (*PID 1*) است. این اثبات می‌کند که محفظه دارای درخت پرسه‌ی مستقل خود است. در همین حال، مشاهده‌ی همین پرسه از روی هاست، یک *PID* کاملاً متفاوت و با شماره‌ی بالا را نشان می‌دهد.



```
Terminal Local × Local (2) × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run ubuntu-base-image /bin/bash
Container started with PID 3703. Press Ctrl+C to stop.
bash-5.1# ps -ef
UID      PID  PPID   C STIME TTY          TIME CMD
root      1      0  0 23:40 ?        00:00:00 /bin/bash
root      4      1  0 23:40 ?        00:00:00 ps -ef
bash-5.1#
```

شکل ۱۶: نمایش درخت پرسه‌ی ایزوله با *PID 1* در داخل محفظه.



```
Terminal Local × Local (2) × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ ps -ef | grep "/bin/bash"
nik      3385     3171  0 19:38 pts/1    00:00:00 /bin/bash --rcfile /home/nik/Downloads/clion-2024.3.3
nik      3596     3171  0 19:38 pts/2    00:00:00 /bin/bash --rcfile /home/nik/Downloads/clion-2024.3.3
root     3697     3385  0 19:40 pts/1    00:00:00 sudo ./my_runner run ubuntu-base-image /bin/bash
root     3698     3697  0 19:40 pts/3    00:00:00 sudo ./my_runner run ubuntu-base-image /bin/bash
root     3699     3698  0 19:40 pts/3    00:00:00 ./my_runner run ubuntu-base-image /bin/bash
root     3703     3699  0 19:40 pts/3    00:00:00 /bin/bash
nik      3777     3596  0 19:41 pts/2    00:00:00 grep --color=auto /bin/bash
nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۱۷: نمایش درخت پرسه‌ی ایزوله در داخل هاست.

مرحله دوم: بررسی Namespace UTS (ایزولهسازی نام میزبان)
نام میزبان را در داخل محفظه و سپس روی هاست بررسی می‌کنیم.

```
# Command inside the container
hostname
```

نتیجه: خروجی در داخل محفظه، "container" است، در حالی که اجرای همین دستور روی هاست، نام واقعی ماشین میزبان را نمایش می‌دهد.

The screenshot shows two terminal windows. The top window is titled 'Terminal' and displays the following commands and output:

```

nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run ubuntu-base-image /bin/bash
Container started with PID 3703. Press Ctrl+C to stop.
bash-5.1# ps -ef
UID      PID  PPID  C STIME TTY          TIME CMD
root        1      0  0 23:40 ?        00:00:00 /bin/bash
root        4      1  0 23:40 ?        00:00:00 ps -ef
bash-5.1# hostname
container
bash-5.1#

```

The bottom window is also titled 'Terminal' and shows:

```

nik@nik-virtual-machine:~/CLionProjects/os$ hostname
nik-virtual-machine
nik@nik-virtual-machine:~/CLionProjects/os$

```

شکل ۱۸: بررسی نام میزبان در محفظه و هاست

مرحله سوم: بررسی Namespace User (ایزوله سازی کاربران) هویت کاربر را در داخل محفظه بررسی کرده و سپس همین دستور را در هاست تکرار می کنیم.

`whoami`

نتیجه: در داخل محفظه، کاربر به عنوان `root` شناخته می شود که با نام عادی کاربر خارج از محفظه و روی هاست، تفاوت دارد.

The screenshot shows two terminal windows. The top window is titled 'Terminal' and displays:

```

nik@nik-virtual-machine:~/CLionProjects/os$ whoami
nik
nik@nik-virtual-machine:~/CLionProjects/os$ 

```

The bottom window is titled 'bash-5.1#' and displays:

```

bash-5.1# whoami
root
bash-5.1#

```

شکل ۱۹: تأیید نگاشت کاربر `root` محفظه به کاربر عادی هاست.

مرحله چهارم: بررسی Namespace Network (ایزوله سازی شبکه) رابطهای شبکه را در داخل محفظه لیست می کنیم.

```
# Command inside the container
ip addr
```

نتیجه: خروجی تنها شامل رابط شبکه‌ی محلی (`lo`) است و هیچ‌کدام از رابطهای فیزیکی یا مجازی هاست قابل مشاهده نیستند، که نشان‌دهنده وجود یک پشتی شبکه‌ی کاملاً ایزوله است.

```

Terminal Local × Local (2) × + ▾
bash-5.1# ps -ef
UID      PID  PPID  C STIME TTY          TIME CMD
root      1      0  0 23:40 ?        00:00:00 /bin/bash
root      4      1  0 23:40 ?        00:00:00 ps -ef
bash-5.1# hostname
container
bash-5.1# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever

Terminal Local × Local (2) × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:0c:29:e4:b4:b6 brd ff:ff:ff:ff:ff:ff
    altname enp2s1
    inet 192.168.44.130/24 brd 192.168.44.255 scope global dynamic noprefixroute ens33
        valid_lft 1306sec preferred_lft 1306sec
    inet6 fe80::84d6:5795:7d04:557f/64 scope link noprefixroute
        valid_lft forever preferred_lft forever

```

شکل ۲۰: رابطهای شبکه در فضای محفظه (بالا) و هاست (پایین)

مرحله پنجم: بررسی Namespace Mount (ایزوله‌سازی نقاط مانت)

نقاط مانت قابل مشاهده در داخل محفظه را لیست می‌کنیم. هدف این است که نشان دهیم محفظه دیدگاه کاملاً ایزوله‌ای از ساختار فایل‌سیستم‌های مانت شده دارد و به نقاط مانت سیستم میزبان دسترسی ندارد.

```
# Command inside the container
mount | grep -E 'overlay|proc'
```

نتیجه: خروجی این دستور، همانظور که در تصویر زیر مشاهده می‌شود، بسیار حداقلی است. این خروجی تنها دو نقطه‌ی مانت اصلی را نشان می‌دهد: اولی، فایل‌سیستم `overlay` است که به عنوان ریشه‌ی اصلی (/) برای محفظه عمل می‌کند. دومی، فایل‌سیستم مجازی `proc`/ است که توسط خود محفظه پس از `chroot` مانت شده است. این لیست کوتاه در تضاد کامل با لیست طولانی نقاط مانت روی سیستم میزبان (که شامل مواردی مانند `/sys`، `/dev`، `/boot` و پارتیشن‌های اصلی دیسک است) قرار دارد. این تفاوت به وضوح اثبات می‌کند که *Mount Namespace* با موفقیت یک دیدگاه ایزوله و تمیز از ساختار مانت سیستم برای محفظه ایجاد کرده است.

```

bash-5.1# mount | grep -E 'overlay|proc'
overlay on / type overlay (rw,relatime,lowerdir=ubuntu-base-image,upperdir=overlay_layers/620/upper,workdir=overlay_layers/620/work,uuid=on,nouserxattr)
proc on /proc type proc (rw,relatime)
bash-5.1#

```

شکل ۲۱: نمایش نقاط مانت ایزوله در داخل محفظه. تنها `overlay` و `proc` قابل مشاهده هستند.

مرحله ششم: بررسی Namespace IPC (ایزوله‌سازی ارتباطات بین فرآیندی) ابدا روی هاست یک صفحه پیام (message queue) ایجاد می‌کنیم.

```
# Command on the host  
ipcmk -Q
```

The screenshot shows a terminal window titled "Terminal". The command "ipcmk -Q" is run, creating a message queue with id 0. Then, "ipcs -q" is run to list all message queues, showing one entry:

key	msqid	owner	perms	used-bytes	messages
0xd78f2252	0	nik	644	0	0

شکل ۲۲: ساخت یک صفحه پیام داخل هاست و مشاهده آن

سپس از داخل محفظه، تلاش می‌کنیم تا صفحه‌های پیام موجود را لیست کنیم.

```
# Command inside the container  
ipcs -q
```

نتیجه: هیچ صفحه‌ای در داخل محفظه یافت نمی‌شود. این نشان می‌دهد که محفظه به منابع IPC هاست دسترسی ندارد.

The screenshot shows a terminal window titled "Terminal". The command "ipcs -q" is run, listing shared memory segments and semaphore arrays, but no message queues are present. The output is as follows:

key	msqid	owner	perms	used-bytes	messages
-----	-------	-------	-------	------------	----------

----- Shared Memory Segments -----
key shmid owner perms bytes nattch status

----- Semaphore Arrays -----
key semid owner perms nsems

bash-5.1# ipcs -q

----- Message Queues -----
key msqid owner perms used-bytes messages

bash-5.1#

شکل ۲۳: صفحه پیام از داخل محفظه دیده نمی‌شد

نتیجه گیری آزمایش این مجموعه آزمایش‌ها به صورت جامع نشان دادند که سیستم مدیریت محفظه به درستی از تمام namespace‌های پیاده‌سازی شده برای ایجاد یک محیط کاملاً ایزوله و امن بهره می‌برد. هر محفظه در یک حباب مجازی از منابع سیستم عمل می‌کند که از هاست و سایر محفظه‌ها جدا شده است.

۵.۳.۳ آزمایش ۵: مدیریت منابع با cgroups

هدف هدف این تست، تأیید این است که سیستم مدیریت محفوظه می‌تواند با استفاده از رابط *cgroups v2*، محدودیت‌های منابع را به درستی بر روی محفوظه‌ها اعمال کند. این قابلیت برای جلوگیری از مصرف بی‌رویه‌ی منابع توسط یک محفوظه و تضمین پایداری کل سیستم، حیاتی است. در این آزمایش، صحبت عملکرد محدودیت‌ها برای سه منبع کلیدی بررسی می‌شود: حافظه، *CPU*، و ورودی/خروجی دیسک (*I/O*).

پیش‌نیازها: فعال‌سازی کنترلرهای **cgroup** برای عملکرد صحیح این تست‌ها، کنترلرهای منابع مورد نیاز باید در سطح ریشه‌ی *cgroups* فعال شده باشند. اسکریپت *setup_rootfs.sh* این کار را به صورت خودکار انجام می‌دهد، اما برای اطمینان می‌توان دستور زیر را به صورت دستی اجرا کرد:

```
echo "+cpu +io +memory +pids" | sudo tee  
/sys/fs/cgroup/cgroup.subtree_control
```

مراحل اجرا و تأیید صحت مرحله اول: تست محدودیت حافظه (–mem) یک محفوظه با محدودیت حافظه‌ی ۵۰ مگابایت اجرا می‌کنیم و به آن دستور می‌دهیم تا با استفاده از ابزار *stress*، ۱۰۰ مگابایت حافظه تخصیص دهد.

```
sudo ./my_runner run --mem 50M ubuntu-base-image /usr/bin/stress --  
vm 1 --vm-bytes 100M
```

نتیجه: قبل از اینکه پروسه‌ی *stress* بتواند کار خود را تمام کند، توسط مکانیزم (*Out – Of – Memory Killer*) هسته‌ی لینوکس کشته می‌شود. برای تأیید این رویداد، لگ‌های هسته را بررسی می‌کنیم.

```
dmesg | grep oom-kill
```

مشاهده‌ی پیغامی مبنی بر کشته شدن پروسه‌ی *stress* به دلیل کمبود حافظه، صحبت عملکرد محدودیت حافظه را اثبات می‌کند.

The terminal window shows the following sequence of commands and output:

```
Terminal Local × Local (2) × + ▾  
nik@nik-virtual-machine:~/ClionProjects/os$ sudo ./my_runner run --mem 50M ubuntu-base-image /usr/bin/stress --vm 1 --vm-bytes 100M  
Container started with PID 4334. Press Ctrl+C to stop.  
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd  
stress: FAIL: [1] (416) <- worker 4 got signal 9  
stress: WARN: [1] (418) now reaping child worker processes  
stress: FAIL: [1] (422) Kill error: No such process  
stress: FAIL: [1] (452) failed run completed in 0s  
Container 4334 has exited. Use 'rm' to clean up.  
  
Terminal Local × Local (2) × + ▾  
nik@nik-virtual-machine:~/ClionProjects/os$ sudo dmesg | grep -i "oom"  
[ 597.985414] stress invoked oom-killer: gfp_mask=0xcc0(GFP_KERNEL), order=0, oom_score_adj=0  
[ 597.985446] oom_kill_process=0x18/0x200  
[ 597.985591] [ pid ] uid tgid total_vm rss rss_anon rss_file rss_shmem pgtables_bytes swapents oom_score_adj name  
[ 597.985598] oom-kill:constraint=CONSTRAINT_MEMORY,nodemask=(null),cpuset=/,mems_allowed=0,oom_memcg=/my_runtime/container_4334,task=memcg_killer,pid=4340,uid=0  
[ 597.985607] Memory cgroup out of memory: Killed process 4340 (stress) total-vm:106112kB, anon-rss:50688kB, file-rss:128kB, shmem-rss:0kB, VIO:0 pgtables:156kB oom_score_adj=0  
nik@nik-virtual-machine:~/ClionProjects/os$
```

شکل ۲۴: تأیید خاتمه‌ی پروسه توسط *OOMKiller* به دلیل عبور از محدودیت حافظه.

در حالی که اگر حافظه مورد استفاده از مقدار ماکریم بیشتر نباشد، محفوظه به درستی شروع به کار می‌کند و با خطای رویه‌رو نخواهیم شد:

The terminal window shows the following sequence of commands and output:

```
Terminal Local × Local (2) × + ▾  
nik@nik-virtual-machine:~/ClionProjects/os$ sudo ./my_runner run --mem 50M ubuntu-base-image /usr/bin/stress --vm 1 --vm-bytes 10M  
Container started with PID 4588. Press Ctrl+C to stop.  
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
```

شکل ۲۵: اجرای محفوظه بدون خطأ

```

Terminal Local × Local (2) × + ▾
top - 21:12:45 up 13 min, 2 users, load average: 0.99, 0.69, 0.55
Tasks: 397 total, 2 running, 395 sleeping, 0 stopped, 0 zombie
%CPU(s): 33.1 us, 3.4 sy, 0.6 ni, 62.1 id, 0.5 wa, 0.0 hi, 0.3 si, 0.0 st
MiB Mem : 3868.5 total, 154.2 free, 2804.2 used, 910.1 buff/cache
MiB Swap: 2680.0 total, 1382.5 free, 1297.5 used, 751.9 avail Mem

      PID USER      PR  NI    VIRT    RES   SHR S %CPU %MEM     TIME+ COMMAND
 4594 root      20   0 13948 10496 256 R 99.3  0.3  1:09.09 stress
2937 nik       20   0 8633012 1.6g 335500 S 27.9 42.8  4:15.23 java
1199 nik       20   0 4568392 174260 105640 S 13.0  4.4  0:40.88 gnome-shell
2983 nik       20   0 235208 23996 19460 S 5.6  0.6  0:09.57 Xwayland
4649 nik       39  19 472960 31780 25688 S 2.7  0.8  0:00.08 tracker-extract
1774 nik       39  19 647896 30216 16720 S 2.0  0.8  0:01.59 tracker-miner-f
1050 nik       9 -11 1946600 16716 15308 S 1.7  0.4  0:01.60 pulseaudio

```

شکل ۲۶: محفظه به درستی اجرا می‌شود

مرحله دوم: تست محدودیت پردازنه (--)cpu

ابتدا یک کار پردازشی سنگین را بدون محدودیت اجرا کرده و زمان آن را به عنوان معیار پایه ثبت می‌کنیم.

```
time sudo ./my_runner run ubuntu-base-image /bin/sh -c "dd if=/dev/zero of=/dev/null bs=1M count=500"
```

```

Terminal Local × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ time sudo ./my_runner run ubuntu-base-image /bin/sh -c "dd if=/dev/zero of=/dev/null bs=1M count=500"
Container started with PID 5201. Press Ctrl+C to stop.
500+0 records in
500+0 records out
524288000 bytes (524 MB, 500 MiB) copied, 0.44031 s, 1.2 GB/s
Container 5201 has exited. Use 'rm' to clean up.

real 0m0.490s
user 0m0.009s
sys 0m0.003s
nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۲۷: دستور پرلود بدون هیچ محدودیتی روی پردازنه

سپس، همان دستور را با محدود کردن توان پردازنه به ۲۰٪ یک هسته (يعني $Quota = 20000$) اجرا می‌کنیم.

```
time sudo ./my_runner run --cpu 20000 ubuntu-base-image /bin/sh -c
"dd if=/dev/zero of=/dev/null bs=1M count=500"
```

نتیجه: زمان اجرای دستور در حالت دوم به شکل قابل توجهی بیشتر از حالت اول (معیار پایه) است. این افزایش زمان به وضوح نشان می‌دهد که توان پردازشی محفظه با موقتیت محدود شده است.

```

Terminal Local × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ time sudo ./my_runner run --cpu 20000 ubuntu-base-image /bin/sh -c "dd if=/dev/zero of=/dev/null bs=1M count=500"
Container started with PID 4859. Press Ctrl+C to stop.
500+0 records in
500+0 records out
524288000 bytes (524 MB, 500 MiB) copied, 2.10645 s, 249 MB/s
Container 4859 has exited. Use 'rm' to clean up.

real 0m2.154s
user 0m0.004s
sys 0m0.009s
nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۲۸: افزایش زمان اجرا به دلیل اعمال محدودیت بر روی CPU

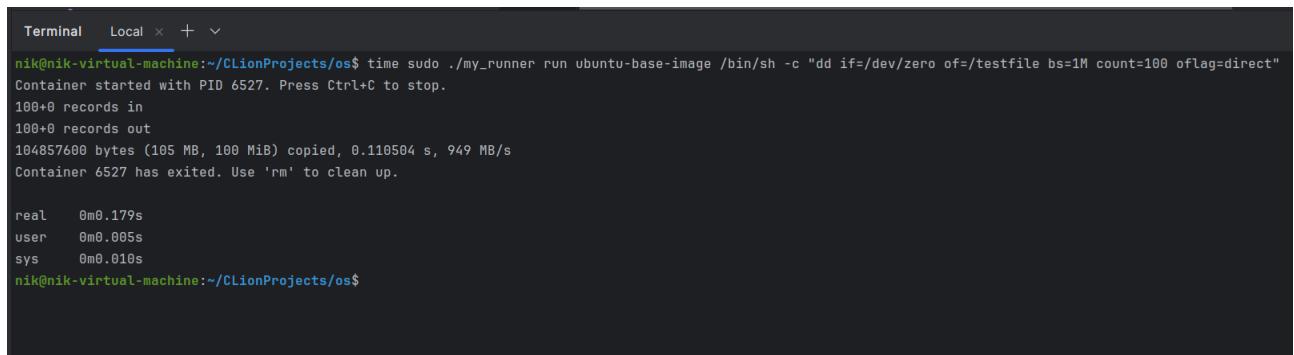
مرحله سوم: تست محدودیت ورودی/خروجی (--)io--.bps

هدف این بخش، تأیید صحت عملکرد محدودیتهای پهنای باند خواندن و نوشتن روی دیسک است. یک چالش کلیدی در این تست، مقابله با مکانیزم کش سیستم عامل system page cache است. به طور پیش فرض، لینوکس عملیات نوشتن را ابتدا در حافظه رم (که بسیار سریع است) انجام داده و سپس در پس زمینه به دیسک منتقل می‌کند. این باعث می‌شود که زمان اجرای دستورات بدون در نظر گرفتن زمان واقعی عملیات دیسک، بسیار کوتاه به نظر برسد. برای به دست آوردن نتایج دقیق، ما باید با استفاده از فلگ‌های iflag=direct (برای خواندن) و oflag=direct (برای نوشتن) در دستور dd، این کش را دور زده و عملیات را مستقیماً روی دیسک انجام دهیم.

آزمایش نوشتن روی دیسک (*Write Test*) گام اول: اجرای تست نوشتن بدون محدودیت. ابتدا، یک فایل ۱۰۰ مگابایتی را با استفاده از *Direct I/O* و بدون هیچ‌گونه محدودیتی در داخل محفظه ایجاد می‌کنیم تا یک معیار پایه برای سرعت نوشتن به دست آوریم.

```
time sudo ./my_runner run ubuntu-base-image \
/bin/sh -c "dd if=/dev/zero of=/testfile bs=1M count=100 oflag=direct"
```

نتیجه: دستور با حداقل سرعت ممکن دیسک اجرا شده و یک زمان پایه ثبت می‌شود.

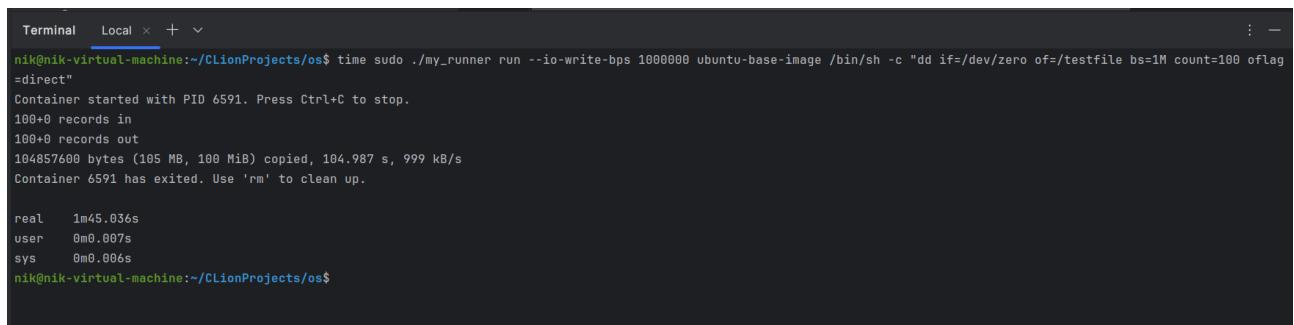


شکل ۲۹: زمان اجرای تست نوشتن روی دیسک بدون اعمال محدودیت.

گام دوم: اجرای تست نوشتن با محدودیت. اکنون همان دستور را با محدود کردن سرعت نوشتن به ۱ مگابایت بر ثانیه اجرا می‌کنیم.

```
time sudo ./my_runner run --io-write-bps 1000000 ubuntu-base-image \
/bin/sh -c "dd if=/dev/zero of=/testfile bs=1M count=100 oflag=direct"
```

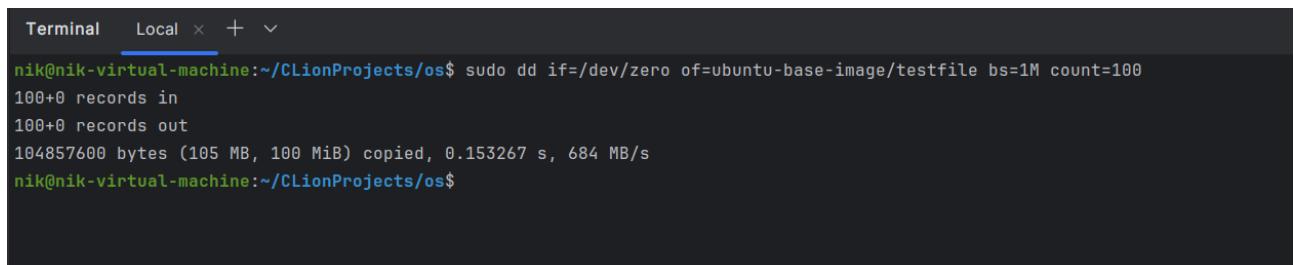
نتیجه: زمان اجرای دستور به شکل چشمگیری افزایش یافته و به حدود ۱۰۰ ثانیه نزدیک می‌شود. این تأخیر طولانی به وضوح نشان می‌دهد که *cgroups I/O* در کنترلر پهنه‌ای باند نوشتن پروسه‌ی محفظه را محدود کرده است.



شکل ۳۰: افزایش شدید زمان اجرا پس از اعمال محدودیت `--io-write-bps`.

آزمایش خواندن از دیسک (*Read Test*) گام اول: آماده‌سازی فایل تست. ابتدا باید یک فایل برای خواندن در داخل ایجاد کنیم. این کار روی هاست انجام می‌شود.

```
dd if=/dev/zero of=ubuntu-base-image/testfile bs=1M count=100
```



شکل ۳۱: ایجاد فایل تست برای خواندن

گام دوم: اجرای تست خواندن بدون محدودیت. یک محفظه اجرا کرده و فایل ۱۰۰ مگابایتی را با استفاده از *I/O Direct* می‌خوانیم تا معیار پایه برای سرعت خواندن به دست آید.

```
time sudo ./my_runner run ubuntu-base-image \
/bin/sh -c "dd if=/testfile of=/dev/null bs=1M iflag=direct"
```

```

Terminal Local + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ time sudo ./my_runner run ubuntu-base-image /bin/sh -c "dd if=/testfile of=/dev/null bs=1M iflag=direct"
Container started with PID 6768. Press Ctrl+C to stop.
100+0 records in
100+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 0.20494 s, 512 MB/s
Container 6768 has exited. Use 'rm' to clean up.

real    0m0.253s
user    0m0.004s
sys     0m0.012s
nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۳۲: زمان اجرای تست خواندن از دیسک بدون اعمال محدودیت

نتیجه: دستور با حداکثر سرعت ممکن دیسک اجرا شده و یک زمان پایه برای عملیات خواندن ثبت می‌شود.
گام سوم: اجرای تست خواندن با محدودیت. همان دستور را با محدود کردن سرعت خواندن به ۲ مگابایت بر ثانیه اجرا می‌کنیم.

```
time sudo ./my_runner run --io-read-bps 2000000 ubuntu-base-image \
/bin/sh -c "dd if=/testfile of=/dev/null bs=1M iflag=direct"
```

نتیجه: زمان اجرای دستور به شدت افزایش یافته و به حدود ۵۰ ثانیه نزدیک می‌شود. این نتیجه نیز به طور قاطع نشان می‌دهد که محدودیت پهنای باند خواندن به درستی توسط cgroup اعمال شده است.

```

Terminal Local + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ time sudo ./my_runner run --io-read-bps 2000000 ubuntu-base-image /bin/sh -c "dd if=/testfile of=/dev/null bs=1M iflag=direct"
Container started with PID 6819. Press Ctrl+C to stop.
100+0 records in
100+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 53.3145 s, 2.0 MB/s
Container 6819 has exited. Use 'rm' to clean up.

real    0m53.363s
user    0m0.002s
sys     0m0.011s
nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۳۳: افزایش شدید زمان اجرا پس از اعمال محدودیت --io-read-bps

نتیجه گیری آزمایش نتایج این آزمایش‌ها نشان می‌دهند که فلگ‌های --mem، --cpu و --io-write-bps به درستی cgroup مربوط به محفظه را پیکربندی می‌کنند. سیستم قادر است به طور موثر مصرف حافظه، توان پردازند و پهنای باند ورودی/خروجی را کنترل کند، که این امر صحت پیاده‌سازی قابلیت مدیریت متایع را تأیید می‌کند.

۶.۳.۳ آزمایش ۶: دستورات CLI و چرخه حیات محفظه

هدف هدف این تست، ارائه یک اعتبارسنجی کامل از چرخه حیات یک محفظه از طریق رابط خط فرمان CLI سیستم است. این آزمایش نشان می‌دهد که برنامه یک سیستم مدیریتی کامل و کارا، مشابه ابزارهایی مانند Podman، برای ایجاد، نظارت، کنترل، و حذف محفظه‌ها فراهم می‌کند. در این تست، دستورات run، start، stop، status، list، rm بررسی می‌شوند.

مراحل اجرا و تأیید صحت برای این تست، از یک پروسه‌ی پردازشی سنگین (stress) استفاده می‌کنیم تا بتوانیم وضعیت "در حال اجرا" بودن محفظه را به صورت بصری با ابزارهایی مانند top مشاهده کنیم.
مرحله اول: ایجاد و اجرای محفظه (run)
یک محفظه در حالت جدا شده detached اجرا می‌کنیم که یک هسته‌ی CPU را به طور کامل اشغال کند.

```
sudo ./my_runner run --detach ubuntu-base-image /usr/bin/stress -c 1
```

نتیجه: دستور، PID محفظه‌ی جدید (مثلًا <PID1>) را چاپ کرده و خارج می‌شود.

```

Terminal Local + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run --detach ubuntu-base-image /usr/bin/stress -c 1
Container started with PID 24079
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner list
CONTAINER PID STATUS COMMAND
24079 Running /usr/bin/stress -c 1
nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۳۴: ساخت یک محفوظه جدید

مرحله دوم: لیست کردن و بررسی وضعیت (status و list) ابتدا از دستور لیست استفاده می‌کنیم تا همه محفظه‌ها را مشاهده کنیم. نتیجه در شکل ۳۴ قابل مشاهده است. سپس وضعیت دقیق محفظه‌ی جدید را بررسی می‌کنیم.

```
sudo ./my_runner list
sudo ./my_runner status <PID1>
```

نتیجه: دستور list محفظه را با وضعیت "Running" نشان می‌دهد. دستور status نیز نشان می‌دهد که زمان مصرف CPU در حال افزایش است.

```
Terminal Local × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner status 24079
--- Status for Container PID 24079 ---
Command : /usr/bin/stress -c 1

--- Resources ---
Memory Usage : 2.72 MB
Total CPU Time : 56.97 seconds
Active Processes/Threads : 2

-----
nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۳۵: دستور status

برای تأیید بصری، از دستور top استفاده می‌کنیم.

```
top -p <PID1>
```

خروجی top نشان می‌دهد که پروسهٔ stress نزدیک به ۱۰۰٪ یک هستهٔ CPU را مصرف می‌کند.

```
Terminal Local × + ▾
top - 18:27:05 up 8:33, 1 user, load average: 1.52, 1.19, 0.71
Tasks: 389 total, 2 running, 387 sleeping, 0 stopped, 0 zombie
%Cpu(s): 29.7 us, 3.1 sy, 0.0 ni, 67.1 id, 0.0 wa, 0.0 hi, 0.1 si, 0.0 st
MiB Mem : 3868.5 total, 149.3 free, 3208.5 used, 510.7 buff/cache
MiB Swap: 2680.0 total, 257.3 free, 2422.6 used. 340.6 avail Mem

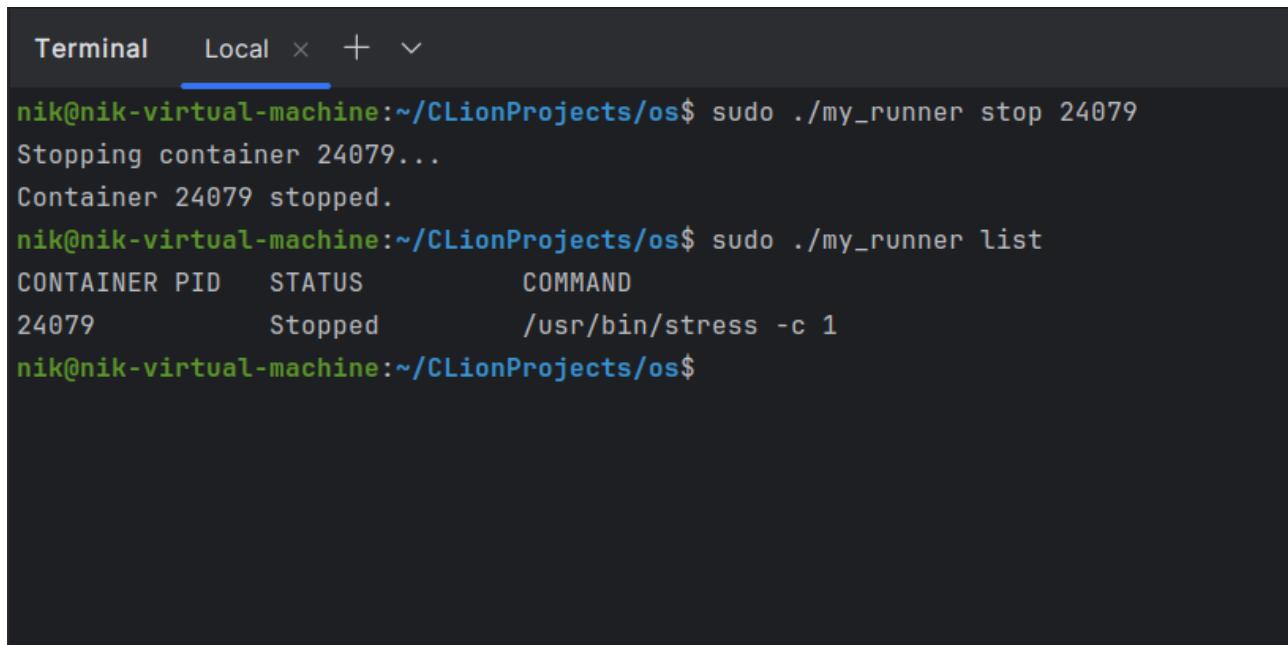
      PID USER      PR  NI    VIRT    RES   SHR S %CPU %MEM TIME+ COMMAND
  24082 root      20   0  3708   256   256 R 99.3  0.0  3:16.63 stress
    3023 nik      20   0 8750368   1.5g 29788 S 24.2 39.1 195:06.74 java
    1181 nik      20   0 4736108 183744 72016 S  2.0  4.6 19:47.52 gnome-shel
  24272 nik      20   0 13348   4224   3328 R  1.6  0.1  0:00.54 top
    467 root     -51   0      0      0      0 S  1.0  0.0  1:48.21 irq/16-vmw
    3068 nik      20   0 243436 10904   5784 S  1.0  0.3  7:00.79 Xwayland
```

شکل ۳۶: تأیید فعالیت محفظه با مشاهدهٔ مصرف بالای CPU.

مرحله سوم: متوقف کردن محفظه (stop) محفوظه‌ی در حال اجرا را با استفاده از PID آن متوقف می‌کنیم.

```
sudo ./my_runner stop <PID1>
```

نتیجه: پیام توقف چاپ می‌شود. اجرای مجدد top نشان می‌دهد که پروسه دیگر وجود ندارد و دستور list وضعیت محفظه را "Stopped" نمایش می‌دهد.



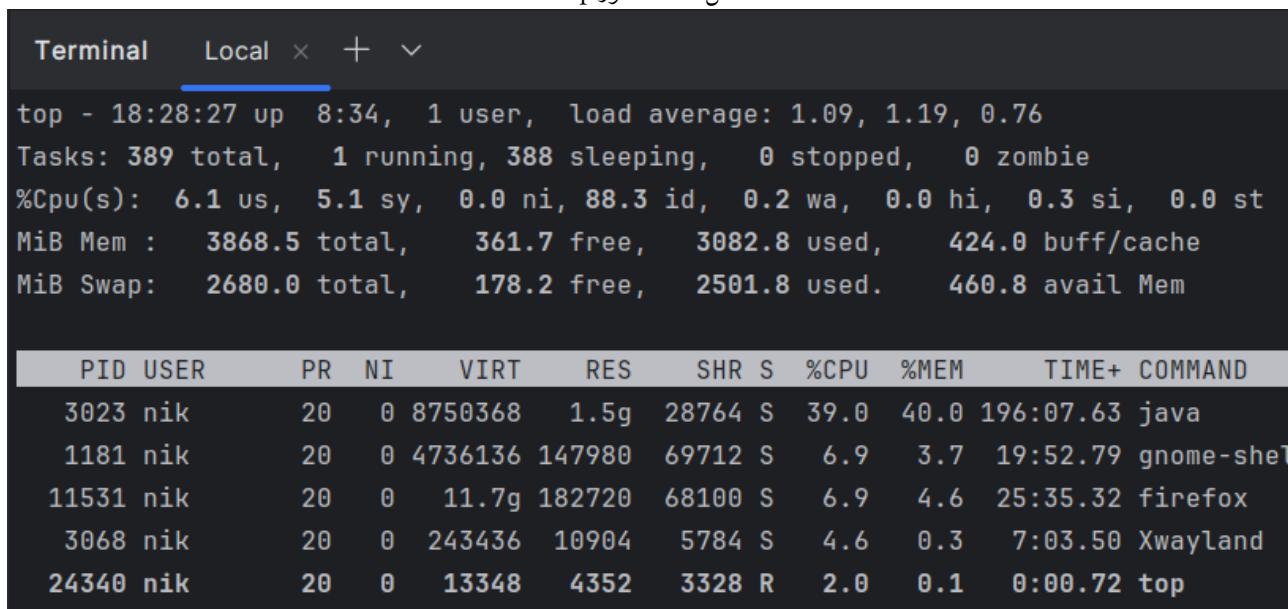
```
Terminal Local × + ▾

nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner stop 24079
Stopping container 24079...
Container 24079 stopped.

nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner list
CONTAINER PID STATUS COMMAND
24079 Stopped /usr/bin/stress -c 1

nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۳۷: دستور stop



```
Terminal Local × + ▾

top - 18:28:27 up 8:34, 1 user, load average: 1.09, 1.19, 0.76
Tasks: 389 total, 1 running, 388 sleeping, 0 stopped, 0 zombie
%Cpu(s): 6.1 us, 5.1 sy, 0.0 ni, 88.3 id, 0.2 wa, 0.0 hi, 0.3 si, 0.0 st
MiB Mem : 3868.5 total, 361.7 free, 3082.8 used, 424.0 buff/cache
MiB Swap: 2680.0 total, 178.2 free, 2501.8 used. 460.8 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
3023 nik 20 0 8750368 1.5g 28764 S 39.0 40.0 196:07.63 java
1181 nik 20 0 4736136 147980 69712 S 6.9 3.7 19:52.79 gnome-shel
11531 nik 20 0 11.7g 182720 68100 S 6.9 4.6 25:35.32 firefox
3068 nik 20 0 243436 10904 5784 S 4.6 0.3 7:03.50 Xwayland
24340 nik 20 0 13348 4352 3328 R 2.0 0.1 0:00.72 top
```

شکل ۳۸: پس از توقف محفظه پردازه درون آن kill می‌شود

مرحله چهارم: راه اندازی مجدد محفظه (start)
محفظه‌ی متوقف شده را با استفاده از PID اصلی آن مجدد راه اندازی می‌کنیم.

```
sudo ./my_runner start <PID1>
```

نتیجه: دستور اعلام می‌کند که محفظه با یک PID جدید (مثلًا <PID2>) راه اندازی شده است. با اجرای مجدد top، مشاهده می‌کنیم که پروسه stress جدید دوباره در حال مصرف ۱۰۰٪ از CPU است.

```

Terminal Local × + ▾

nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner start 24079
Starting container 24079...
Container 24079 started with new PID 24373
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner list
CONTAINER PID STATUS COMMAND
24373 Running /usr/bin/stress -c 1
nik@nik-virtual-machine:~/CLionProjects/os$

```

شکل ۳۹: تأیید راهاندازی مجدد محفظه و فعالیت دوباره‌ی پروسه‌ی stress

```

Terminal Local × + ▾

top - 18:29:23 up 8:35, 1 user, load average: 1.00, 1.13, 0.76
Tasks: 375 total, 2 running, 373 sleeping, 0 stopped, 0 zombie
%Cpu(s): 28.2 us, 2.1 sy, 0.0 ni, 69.6 id, 0.1 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 3868.5 total, 342.8 free, 3078.7 used, 447.0 buff/cache
MiB Swap: 2680.0 total, 180.0 free, 2500.0 used. 464.8 avail Mem

          PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
24379 root      20   0  3708   256   256 R 99.7  0.0  0:25.00 stress
3023 nik       20   0 8750368  1.5g 28764 S 18.4 40.0 196:26.00 java
1181 nik       20   0 4736256 147980 69712 S  1.3  3.7 19:56.09 gnome-shell
23741 root      20   0      0      0      0 I  1.3  0.0  0:18.05 kworker/3:
24394 nik       20   0 13348   4224   3328 R  1.3  0.1  0:00.28 top

```

شکل ۴۰: پردازه دوباره شروع به کار می‌کند

مرحله پنجم: توقف و حذف نهایی (rm و stop) در نهایت، محفظه را با *PID* جدید آن متوقف کرده و سپس به طور کامل حذف می‌کنیم.

```

sudo ./my_runner stop <PID2>
sudo ./my_runner rm <PID2>

```

نتیجه: پس از اجرای این دستورات، اجرای مجدد sudo ./my_runner list پیغام "No containers exist." را نمایش می‌دهد که به معنای پاکسازی کامل منابع محفظه است.

```

Terminal Local × + ▾

nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner stop 24373
Stopping container 24373...
Container 24373 stopped.
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner rm 24373
Removing container 24373...
Container 24373 removed.
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner list
No containers exist.

```

شکل ۴۱: حذف محفظه

نتیجه گیری آزمایش این آزمایش نشان داد که رابط خط فرمان برنامه، یک چرخه‌ی حیات کامل و منسجم را برای مدیریت محفظه‌ها فراهم می‌کند. قابلیت‌های run، start، stop، status، list، rm همگی به درستی کار کرده و به کاربر اجازه می‌دهند تا به صورت کارآمد، محفظه‌ها را ایجاد، نظارت و مدیریت کنند. این امر، یکی از اهداف اصلی پروژه را در ایجاد یک ابزار مدیریتی کاربردی محقق می‌سازد.

۷.۳.۳ آزمایش ۷: توقف موقت با `freeze` و `thaw`

هدف هدف این تست، تأیید صحت عملکرد دستورات `freeze` و `thaw` است. این قابلیت با استفاده از کنترلر `cgroups` در `freezer` می‌باشد و به کاربر اجازه می‌دهد تا تمام پروسه‌های درون یک محفظه را به صورت موقت متوقف (*suspend*) کرده و در زمان دلخواه، اجرای آنها را دقیقاً از همان نقطه‌ای که متوقف شده بودند، از سر بگیرد. این آزمایش تفاوت اساسی این قابلیت با چرخه `stop/start` را نشان می‌دهد که در آن، وضعیت داخلی پروسه از بین می‌رود.

مراحل اجرا و تأیید صحت برای این تست، از یک اسکریپت شمارنده استفاده می‌کنیم تا بتوانیم حفظ شدن وضعیت داخلی پروسه (مقدار متغیر شمارنده) را پس از `thaw` به وضوح مشاهده کنیم.

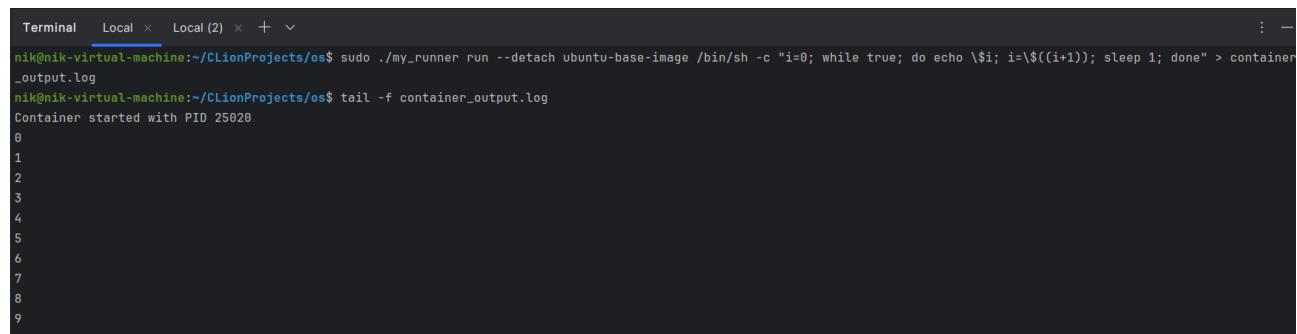
مرحله اول: اجرای محفظه و مشاهده خروجی زنده
یک محفظه در حالت جدا شده `detached` اجرا کرده و خروجی استاندارد آن را به یک فایل لاغ به نام `container_output.log` هدایت می‌کنیم.

```
sudo ./my_runner run --detach ubuntu-base-image \
/bin/sh -c "i=0; while true; do echo \$i; i=\$((i+1)); sleep 1;
done" > container_output.log
```

سپس با استفاده از دستور `tail -f`، خروجی فایل لاغ را به صورت زنده در یک ترمینال دیگر مشاهده می‌کنیم.

```
tail -f container_output.log
```

نتیجه: اعداد '۰، ۱، ۲، ۳، ...' به ترتیب و با فاصله‌ی یک ثانیه در ترمینال نمایش داده می‌شوند.



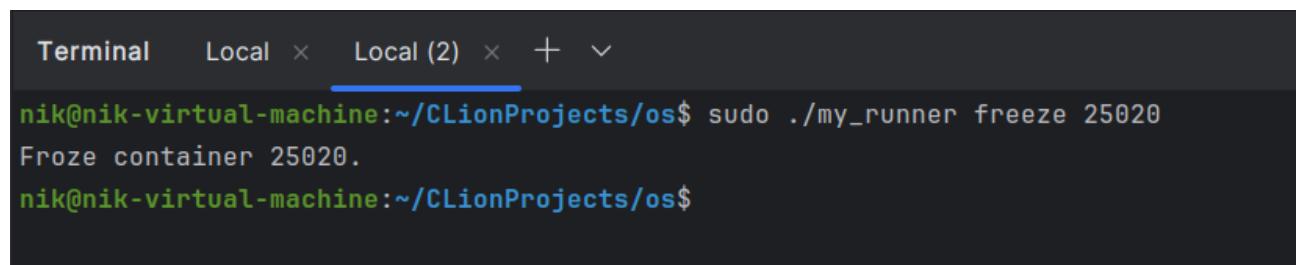
شکل ۴۲: اجرای یک محفظه شمارنده

مرحله دوم: متوقف کردن موقت محفظه (`freeze`)
پس از چند ثانیه، با استفاده از *PID* محفظه، دستور `freeze` را اجرا می‌کنیم.

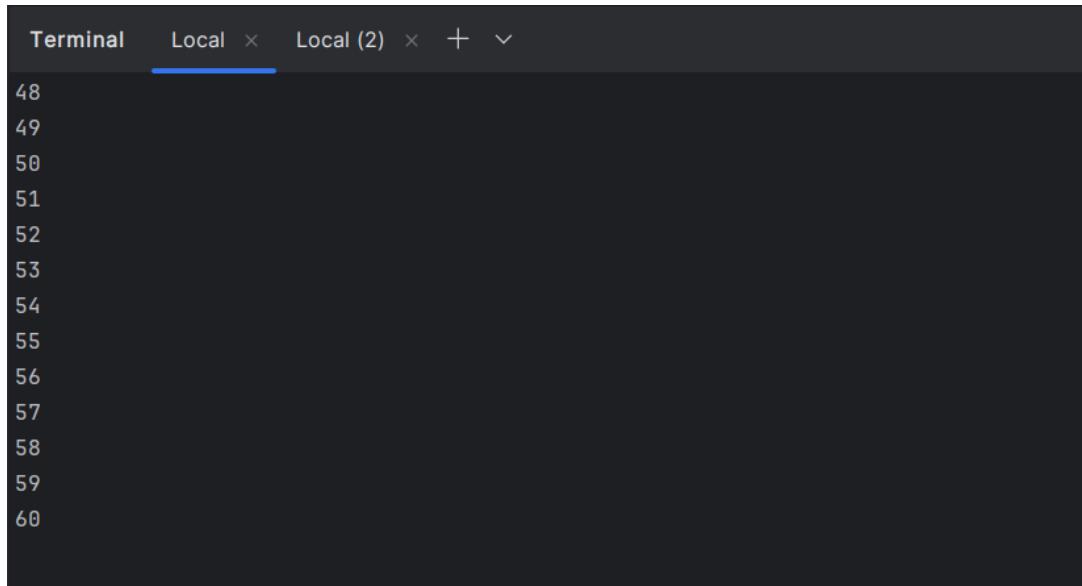
```
sudo ./my_runner freeze <PID>
```

نتیجه: چاپ اعداد در ترمینالی که `tail` در آن در حال اجراست، بلا فاصله متوقف می‌شود. همچنین با اجرای دستور `status`، وضعیت محفوظه

"Frozen" نمایش داده می‌شود.



شکل ۴۳: توقف اجرای پروسه‌ی شمارنده پس از اجرای دستور `freeze`.



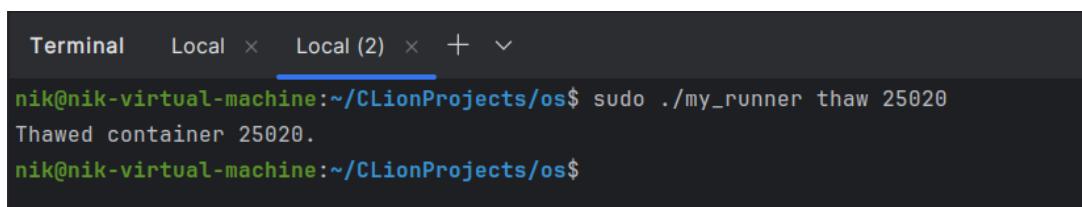
```
48
49
50
51
52
53
54
55
56
57
58
59
60
```

شکل ۴۴: شمارش متوقف می‌شود.

مرحله سوم: از سرگیری اجرای محفظه (thaw) پس از چند ثانیه توقف، دستور thaw را اجرا می‌کنیم.

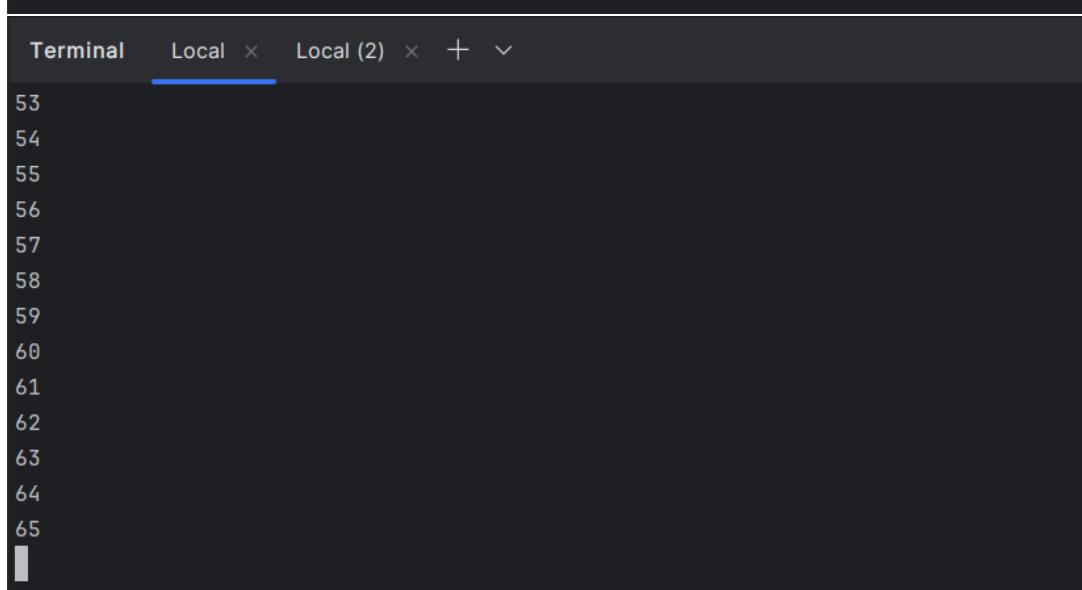
```
sudo ./my_runner thaw <PID>
```

نتیجه: نتیجه کلیدی: شمارش اعداد دقیقاً از همان عددی که متوقف شده بود، ادامه می‌یابد (اینجا اگر روی '۶۰' متوقف شده بود، درنتیجه با '۶۱، ۶۲، ۶۳، ...' ادامه می‌دهد). این نتیجه به طور قطعی ثابت می‌کند که وضعیت داخلی پروسه (مقدار متغیر 'i') در طول مدت توقف، در حافظه حفظ شده است.



```
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner thaw 25020
Thawed container 25020.

nik@nik-virtual-machine:~/CLionProjects/os$
```



```
53
54
55
56
57
58
59
60
61
62
63
64
65
```

شکل ۴۵: از سرگیری اجرای پروسه و حفظ شدن وضعیت داخلی آن.

نتیجه‌گیری آزمایش این آزمایش به وضوح نشان داد که مکانیزم freeze و thaw به درستی با استفاده از کنترلر freezer در cgroups پیاده‌سازی شده است. این قابلیت، امکان توقف موقت و از سرگیری اجرای محفظه‌ها را بدون از دست دادن وضعیت داخلی آن‌ها فراهم می‌کند که این یک تفاوت بنیادی با چرخه stop/start است و کاربردهای مهمی در مدیریت پیشرفته‌ی وضعیت برنامه‌ها دارد.

۸.۳.۳ آزمایش ۸: ارتباط بین محفظه‌ها با اشتراک‌گذاری فضای نام IPC

هدف هدف این تست، تأیید صحت عملکرد فلگ `--share-ipc` است. به طور پیش‌فرض، هر محفظه در یک فضای نام ارتباطات بین فرآیندی (*IPC Namespace*) کاملاً ایزوله ایجاد می‌شود. این ایزوله‌سازی از دسترسی محفظه‌ها به منابع *IPC* یکدیگر (مانند حافظه اشتراکی، سمافورها و صفحه‌های پیام) جلوگیری می‌کند. این آزمایش نشان می‌دهد که چگونه می‌توان با استفاده از این فلگ، این رفتار پیش‌فرض را تغییر داده و به دو محفظه‌ی مجزا اجازه داد تا با یکدیگر ارتباط برقرار کنند.

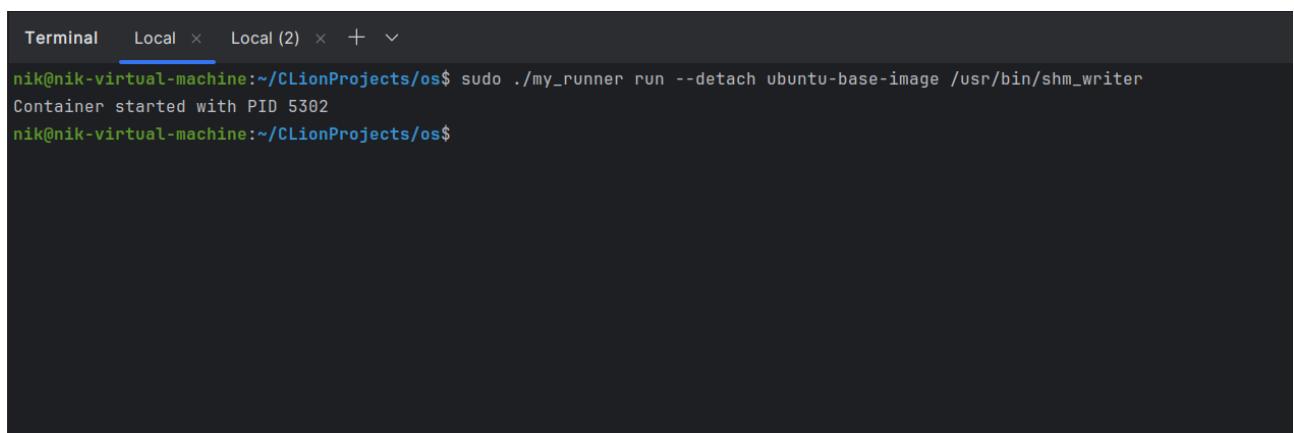
پیش‌نیازها: برنامه‌های تست برای این تست، از دو برنامه‌ی ساده به نام‌های `shm_reader` و `shm_writer` استفاده می‌شود که توسط اسکریپت `setup_rootfs.sh` کامپایل و در اینمیج پایه کپی شده‌اند.

● **shm_writer**: یک قطعه حافظه‌ی اشتراکی با یک کلید مشخص ایجاد کرده، پیامی در آن می‌نویسد و منتظر می‌ماند.

● **shm_reader**: تلاش می‌کند به همان قطعه حافظه‌ی اشتراکی متصل شده، پیام را بخواند، چاپ کند و سپس آن را از سیستم حذف نماید.

مراحل اجرا و تأیید صحت مرحله اول: بررسی ایزوله‌سازی پیش‌فرض (حالت کنترل) ابتدا نشان می‌دهیم که در حالت عادی، ارتباط بین دو محفظه ممکن نیست. یک محفظه برای `shm_writer` بدون فلگ `--share-ipc` اجرا می‌کنیم.

```
sudo ./my_runner run --detach ubuntu-base-image /usr/bin/shm_writer
```



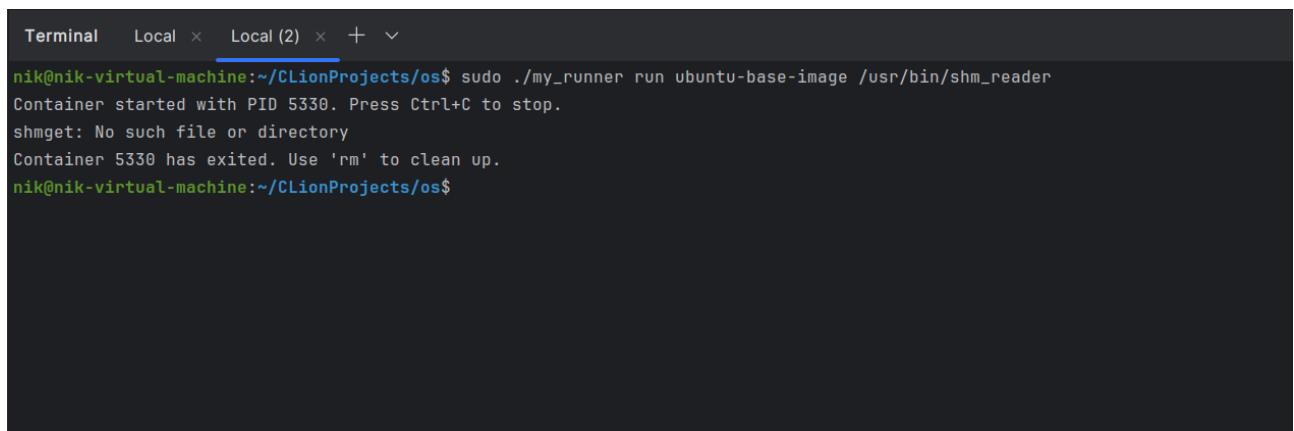
The screenshot shows a terminal window with the title 'Terminal'. It displays the command: `nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run --detach ubuntu-base-image /usr/bin/shm_writer`. Below the command, the output shows: 'Container started with PID 5302'. The prompt then changes to `nik@nik-virtual-machine:~/CLionProjects/os$`.

شکل ۴۶: اجرای `shm_writer`

سپس، تلاش می‌کنیم تا محفظه‌ی `shm_reader` را نیز در حالت پیش‌فرض اجرا کنیم.

```
sudo ./my_runner run ubuntu-base-image /usr/bin/shm_reader
```

نتیجه: برنامه‌ی `shm_reader` با شکست مواجه شده و خطای `shmget : No such file or directory` را نمایش می‌دهد. این خطأ به درستی نشان می‌دهد که دو محفظه در فضاهای نام *IPC* جداگانه قرار دارند و قادر به یافتن منابع یکدیگر نیستند.



The screenshot shows a terminal window with the title 'Terminal'. It displays the command: `nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run ubuntu-base-image /usr/bin/shm_reader`. Below the command, the output shows: 'Container started with PID 5330. Press Ctrl+C to stop.' Then it shows the error message: 'shmget: No such file or directory'. Finally, it says: 'Container 5330 has exited. Use 'rm' to clean up.' The prompt then changes to `nik@nik-virtual-machine:~/CLionProjects/os$`.

شکل ۴۷: عدم موفقیت در ارتباط به دلیل ایزوله بودن فضاهای نام *IPC*.

مرحله دوم: تست ارتباط با فضای نام اشتراکی

اکنون آزمایش را با استفاده از فلگ `--share-ipc` تکرار می‌کنیم. ابتدا محفظه‌ی نویسنده را با این فلگ اجرا می‌نماییم.

```
sudo ./my_runner run --share-ipc --detach ubuntu-base-image /usr/bin/shm_writer
```

```

Terminal Local × Local (2) × + ∨
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run --share-ipc --detach ubuntu-base-image /usr/bin/shm_writer
Container started with PID 5465
nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۴۸: ایجاد محفظه نویسنده با فلگ `--share-ipc`

سپس، محفظه‌ی خواننده را نیز با همین فلگ اجرا می‌کنیم.

```
sudo ./my_runner run --share-ipc ubuntu-base-image /usr/bin/shm_reader
```

نتیجه: این بار، برنامه‌ی `shm_reader` با موفقیت اجرا شده و پیام "Hello from writer!" را که توسط محفظه‌ی اول نوشته شده بود، می‌خواند و چاپ می‌کند. این موفقیت نشان می‌دهد که هر دو محفظه با پیوستن به فضای نام *IPC* هاست، توانسته‌اند به یک منبع حافظه‌ی اشتراکی مشترک دسترسی پیدا کنند.

```

Terminal Local × Local (2) × + ∨
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run --share-ipc ubuntu-base-image /usr/bin/shm_reader
Container started with PID 5489. Press Ctrl+C to stop.
Read from shared memory: Hello from writer!
Container 5489 has exited. Use 'rm' to clean up.
nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۴۹: ارتباط موفق بین دو محفظه با استفاده از فلگ `--share-ipc`.

نتیجه‌گیری آزمایش نتایج این آزمایش به وضوح نشان می‌دهند که سیستم هم ایزوله‌سازی پیش‌فرض *IPC* را به درستی پیاده‌سازی کرده و هم قابلیت اشتراک‌گذاری این فضا را از طریق فلگ `--share-ipc` به صورت صحیح فراهم نموده است.

۹.۳.۳ آزمایش ۹: انتشار رویدادهای Mount با `--propagate-mount`

هدف هدف این تست، تأیید صحت عملکرد فلگ `--propagate-mount` است. این قابلیت به سیستم اجازه می‌دهد تا یک دایرکتوری روی هاست را به گونه‌ای با محفظه به اشتراک بگذارد که هرگونه عملیات مانند آنمانست جدیدی که در آن دایرکتوری روی هاست رخ دهد، به صورت پویا و بی‌درنگ در داخل محفظه در حال اجرا نیز منعکس شود. این ویژگی برای ستابیوهایی مانند اتصال حافظه‌های جانبی (*storage devices*) به هاست پس از اجرای محفظه، بسیار کاربردی است.

پیش‌نیازها: آماده‌سازی نقطه مانند اشتراکی برای عملکرد صحیح این قابلیت، دایرکتوری مورد نظر روی هاست باید ابتدا به یک نقطه مانند تبدیل شده و سپس نوع انتشار آن به اشتراکی (*shared*) تغییر یابد.

```

# Create the necessary directories on the host
sudo mkdir /mnt/shared
sudo mkdir /mnt/my_device_data

# Turn the directory into a mount point and set it as shared
sudo mount --bind /mnt/shared /mnt/shared
sudo mount --make-shared /mnt/shared
```

```

Terminal Local + ▾
nik@nik-virtual-machine:~/CLionProjects/test$ sudo mkdir /mnt/shared
nik@nik-virtual-machine:~/CLionProjects/test$ sudo mkdir /mnt/my_device_data
nik@nik-virtual-machine:~/CLionProjects/test$



Terminal Local + ▾
nik@nik-virtual-machine:~/CLionProjects/test$ sudo mount --bind /mnt/shared /mnt/shared
nik@nik-virtual-machine:~/CLionProjects/test$ sudo mount --make-shared /mnt/shared
nik@nik-virtual-machine:~/CLionProjects/test$

```

شکل ۵۰: اجرای پیش‌نیازها

مراحل اجرا و تأیید صحت مرحله اول: اجرای محفظه با مانت اشتراکی یک محفظه را با استفاده از فلگ `--propagate-mount` و با مشخص کردن مسیر `/mnt/shared` اجرا می‌کنیم.

```
sudo ./my_runner run --propagate-mount /mnt/shared ubuntu-base-image /bin/bash
```

```

Terminal Local Local (2) + ▾
nik@nik-virtual-machine:~/CLionProjects/test$ sudo ./my_runner run --propagate-mount /mnt/shared ubuntu-base-image /bin/bash
Container started with PID 5646. Press Ctrl+C to stop.
bash-5.1#

```

شکل ۵۱: اجرای محفوظه

سپس در داخل محفظه، محتویات دایرکتوری `/mnt/shared` را لیست می‌کنیم. نتیجه: این دایرکتوری در داخل محفوظه وجود دارد ولی در حال حاضر خالی است.

```

Terminal Local Local (2) + ▾
nik@nik-virtual-machine:~/CLionProjects/test$ sudo ./my_runner run --propagate-mount /mnt/shared ubuntu-base-image /bin/bash
Container started with PID 5646. Press Ctrl+C to stop.
bash-5.1# ls /mnt
shared
bash-5.1#

```

شکل ۵۲: دایرکتوری خالیست

مراحله دوم: ایجاد یک رویداد مانت روی هاست در حالی که محفظه در حال اجراست، در یک ترمینال دیگر روی هاست، یک دایرکتوری جدید در مسیر اشتراکی ایجاد کرده و یک "دستگاه" مجازی (که همان دایرکتوری `/mnt/my_device_data` است) را روی آن مانت می‌کنیم.

```
# Commands on the host
sudo mkdir /mnt/shared/my_device_mount
sudo mount --bind /mnt/my_device_data /mnt/shared/my_device_mount
```

```

Terminal Local × Local (2) × + ▾
nik@nik-virtual-machine:~/CLionProjects/test$ sudo mkdir /mnt/shared/my_device_mount
nik@nik-virtual-machine:~/CLionProjects/test$ sudo mount --bind /mnt/my_device_data /mnt/shared/my_device_mount
nik@nik-virtual-machine:~/CLionProjects/test$ sudo touch /mnt/shared/my_device_mount/hello_from_host.txt
nik@nik-virtual-machine:~/CLionProjects/test$

```

شکل ۵۳: اضافه شدن یک دستگاه جدید و مانت آن در مسیر دایرکتوری هاست

مرحله سوم: تأیید انتشار رویداد در داخل محفظه
اکنون به ترمینال محفظه بازگشته و دوباره محتويات دایرکتوری `/mnt/shared` را لیست می کنیم.

Command inside the container

```
ls /mnt/shared
```

نتیجه: دایرکتوری `my_device_mount` که روی هاست ایجاد و مانت شده بود، اکنون به صورت خودکار در داخل محفظه نیز ظاهر شده است.
این نتیجه به طور قطعی ثابت می کند که رویداد مانت از هاست به محفظه منتشر شده است.

```

Terminal Local × Local (2) × + ▾
nik@nik-virtual-machine:~/CLionProjects/test$ sudo ./my_runner run --propagate-mount /mnt/shared ubuntu-base-image /bin/bash
Container started with PID 5646. Press Ctrl+C to stop.
bash-5.1# ls /mnt
shared
bash-5.1# ls /mnt/shared
my_device_mount
bash-5.1# ls /mnt/shared/my_device_mount
hello_from_host.txt
bash-5.1#

```

شکل ۵۴: مشاهده نقطعی مانت جدید در داخل محفظه پس از انتشار رویداد از هاست.

نتیجه گیری آزمایش این آزمایش نشان داد که قابلیت انتشار رویدادهای مانت به درستی پیاده سازی شده است. این ویژگی، فراتر از یک اشتراک گذاری ساده دایرکتوری عمل کرده و یک ارتباط پویا بین ساختار فایل سیستم هاست و محفظه برقرار می کند. اهمیت این قابلیت در سفاریوهای عملی بسیار زیاد است؛ به عنوان مثال، یک مدیر سیستم می تواند یک حافظه جانبی جدید (مانند یک فلاش درایو یا یک دیسک سخت خارجی) را به سیستم میزبان متصل کرده و آن را در مسیر اشتراکی مانت کند. با استفاده از این قابلیت، یک محفظه سرویس دهنده (مانند یک پایگاه داده) که از قبل در حال اجرا بوده، می تواند بالافاصله و بدون نیاز به راه اندازی مجدد، به این حافظه جدید دسترسی پیدا کرده و از آن برای عملیاتی مانند پشتیبان گیری (*backup*) استفاده کند. این انعطاف پذیری، مدیریت ذخیره سازی پویا را به شکل قابل توجهی ساده و کارآمد می سازد.

۱۰.۳.۳ آزمایش ۱۰: قفل کردن روی هسته CPU و زمانبندی

هدف هدف این تست، تأیید صحت عملکرد فلگ `--pin-cpu` است. این قابلیت پیشرفته برای بهبود عملکرد برنامه های حساس به تأخیر (*latency - sensitive*) طراحی شده و با دو مکانیزم اصلی، تداخل و پرش های زمینه ای (*context switch*) را به حداقل می رساند:

۱. تنظیم وابستگی به CPU (*CPU Affinity*): پروسه هایی محفوظه را به یک هسته های CPU خاص قفل می کند.

۲. تغییر سیاست زمان بندی: سیاست زمان بندی پروسه را به (*SCHED_RR*) تغییر می دهد که یک سیاست قطعی تر و مناسب برای بارهای کاری بی درنگ است.

این آزمایش همچنین نشان می دهد که سیستم چگونه محفظه های پین شده جدید را به صورت نوبت گردشی (*round-robin*) بین هسته های موجود توزیع می کند.

پیش نیازها این تست نیازمند یک سیستم با حداقل چهار هسته های CPU برای نمایش کامل رفتار توزیع است. ابزارهای `chrt` و `taskset` برای تأیید صحت استفاده می شوند. قبل از شروع، فایل وضعیت مربوط به تخصیص CPU را حذف می کنیم تا تخصیص از هسته های صفر آغاز شود.

```
rm -f /tmp/my_runtime_next_cpu
```

البته توجه کنید که این تست را با تعداد هسته های پایین تر می توان انجام داد و تنها تعداد محفظه های در حال اجرا را تغییر داد.

مراحل اجرا و تأیید صحت مرحله اول: بررسی محفظه‌های پین شده چهار محفظه را با فلگ `--pin-cpu`-- اجرا کرده و بررسی می‌کنیم که هر کدام به یک هسته‌ی مجزا (به ترتیب ۰، ۱، ۲، و ۳) اختصاص داده شده و سیاست زمان‌بندی آن‌ها `SCHED_RR` باشد.

```
# Launch four containers with the --pin-cpu flag
sudo ./my_runner run --pin-cpu --detach ubuntu-base-image /bin/sh -c "while true; do :; done"
# Note <PID1>
sudo ./my_runner run --pin-cpu --detach ubuntu-base-image /bin/sh -c "while true; do :; done"
# Note <PID2>
# ... and so on for PID3 and PID4
```

سپس برای هر محفظه، وابستگی به *CPU* و سیاست زمان‌بندی را بررسی می‌کنیم.

```
# Verification for Container 1
taskset -c -p <PID1>
chrt -p <PID1>
```

نتیجه: برای محفظه‌ی اول، `taskset` لیست وابستگی را، و `chrt` سیاست زمان‌بندی را `SCHED_RR` گزارش می‌دهد. این روند برای محفظه‌های بعدی به ترتیب روی هسته‌های ۱، ۲، و ۳ تکرار می‌شود.

```
Terminal Local × Local (2) × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ rm -f /tmp/my_runtime_next_cpu
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run --pin-cpu --detach ubuntu-base-image /bin/sh -c "while true; do :; done"
Container started with PID 6181
nik@nik-virtual-machine:~/CLionProjects/os$ taskset -c -p 6181
pid 6181's current affinity list: 0
nik@nik-virtual-machine:~/CLionProjects/os$ ps -o pid,comm,sched,psr -p 6181
  PID COMMAND      SCHED PSR
  6181 sh          2   0
nik@nik-virtual-machine:~/CLionProjects/os$ chrt -p 6181
pid 6181's current scheduling policy: SCHED_RR
pid 6181's current scheduling priority: 50
nik@nik-virtual-machine:~/CLionProjects/os$
```



```
Terminal Local × Local (2) × Local (3) × Local (4) × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run --pin-cpu --detach ubuntu-base-image /bin/sh -c "while true; do :; done"
Container started with PID 6308
nik@nik-virtual-machine:~/CLionProjects/os$ taskset -c -p 6308
pid 6308's current affinity list: 1
nik@nik-virtual-machine:~/CLionProjects/os$ chrt -p 6308
pid 6308's current scheduling policy: SCHED_RR
pid 6308's current scheduling priority: 50
nik@nik-virtual-machine:~/CLionProjects/os$
```



```
Terminal Local × Local (2) × Local (3) × Local (4) × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run --pin-cpu --detach ubuntu-base-image /bin/sh -c "while true; do :; done"
Container started with PID 6380
nik@nik-virtual-machine:~/CLionProjects/os$ taskset -c -p 6380
pid 6380's current affinity list: 2
nik@nik-virtual-machine:~/CLionProjects/os$ chrt -p 6380
pid 6380's current scheduling policy: SCHED_RR
pid 6380's current scheduling priority: 50
nik@nik-virtual-machine:~/CLionProjects/os$
```



```
Terminal Local × Local (2) × Local (3) × Local (4) × + ▾
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run --pin-cpu --detach ubuntu-base-image /bin/sh -c "while true; do :; done"
Container started with PID 6430
nik@nik-virtual-machine:~/CLionProjects/os$ taskset -c -p 6430
pid 6430's current affinity list: 3
nik@nik-virtual-machine:~/CLionProjects/os$ chrt -p 6430
Command 'chhrt' not found, did you mean:
  command 'chrt' from deb util-linux (2.37.2-4ubuntu3.4)
Try: sudo apt install <deb name>
nik@nik-virtual-machine:~/CLionProjects/os$ chrt -p 6430
pid 6430's current scheduling policy: SCHED_RR
pid 6430's current scheduling priority: 50
nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۵۵: تأیید پین شدن متواالی محفوظه‌ها به هسته‌های مختلف CPU

مرحله دوم: بررسی رفتار پیش‌فرض (محفظه‌ی پین شده)

اگر نیز محفوظه را بدون فلگ `--pin-cpu`-- اجرا می‌کنیم تا رفتار پیش‌فرض سیستم را مشاهده کنیم.

```
sudo ./my_runner run --detach ubuntu-base-image /bin/sh -c "while true; do :; done"
```

نتیجه: با اجرای دستورات تأیید صحت روی این محفوظه جدید:

لیست وابستگی را 0-3 گزارش می‌دهد، به این معنی که زمان‌بند لینوکس آزاد است تا پروسه را روی هر یک از چهار هسته اجرا کند.

● سیاست زمان‌بندی را SCHED_OTHER گزارش می‌دهد که همان سیاست زمان‌بندی استاندارد و اشتراک زمانی لینوکس است.

```
Terminal Local × + ~
nik@nik-virtual-machine:~/CLionProjects/os$ sudo ./my_runner run --detach ubuntu-base-image /bin/sh -c "while true; do :; done"
Container started with PID 6797
nik@nik-virtual-machine:~/CLionProjects/os$ taskset -c -p 6797
pid 6797's current affinity list: 0-3
nik@nik-virtual-machine:~/CLionProjects/os$ chrt -p 6797
pid 6797's current scheduling policy: SCHED_OTHER
pid 6797's current scheduling priority: 0
nik@nik-virtual-machine:~/CLionProjects/os$
```

شکل ۵۶: بررسی ویژگی‌های یک محفظه پین نشده

این نتایج بهوضوح تفاوت بین حالت پیش‌فرض و حالت بهینه‌سازی شده را نشان می‌دهند.

نتیجه گیری آزمایش این آزمایش نشان داد که فلگ pin-cpu -- به درستی وابستگی CPU و سیاست زمان‌بندی پروسه‌ی محفظه را تغییر می‌دهد و همچنین توزیع گردشی بین هسته‌ها را به درستی پیاده‌سازی می‌کند.

۱۱.۳.۱۱ آزمایش ۱۱: نظارت بر فراخوانی‌های سیستمی با eBPF

هدف هدف این تست، تأیید صحت عملکرد ابزار نظارتی monitor.py است که با استفاده از فناوری eBPF پیاده‌سازی شده است. این ابزار برای رهگیری و ثبت وقایع سطح پایین مربوط به ایجاد محفظه طراحی شده است. به طور خاص، این آزمایش نشان می‌دهد که اسکریپت ما قادر است دو نوع فراخوانی سیستمی کلیدی را که توسط برنامه my_runner انجام می‌شود، با موقوفیت شناسایی و لاغ کند:

۱. فراخوانی clone هنگامی که با پرچم‌های ایجاد namespace جدید همراه است.

۲. فراخوانی mkdir هنگامی که برای ایجاد یک دایرکتوری cgroup جدید در مسیر /sys/fs/cgroup به کار می‌رود.

پیش‌نیازها این تست نیازمند دو ترمینال مجزا است. قبل از شروع، فایل لاغ قبلی (ebpf_log.txt) را حذف می‌کنیم تا از صحت نتایج اطمینان حاصل شود.

مراحل اجرا و تأیید صحت مرحله اول: اجرای اسکریپت نظارتی monitor.py را با دسترسی sudo اجرا می‌کنیم. این اسکریپت برنامه‌ی eBPF را در هسته بارگذاری کرده و منتظر دریافت رویدادها می‌ماند.

```
sudo python3 monitor.py
```

نتیجه: اسکریپت پیام ”Starting eBPF monitoring...“ را چاپ کرده و در حالت انتظار باقی می‌ماند.

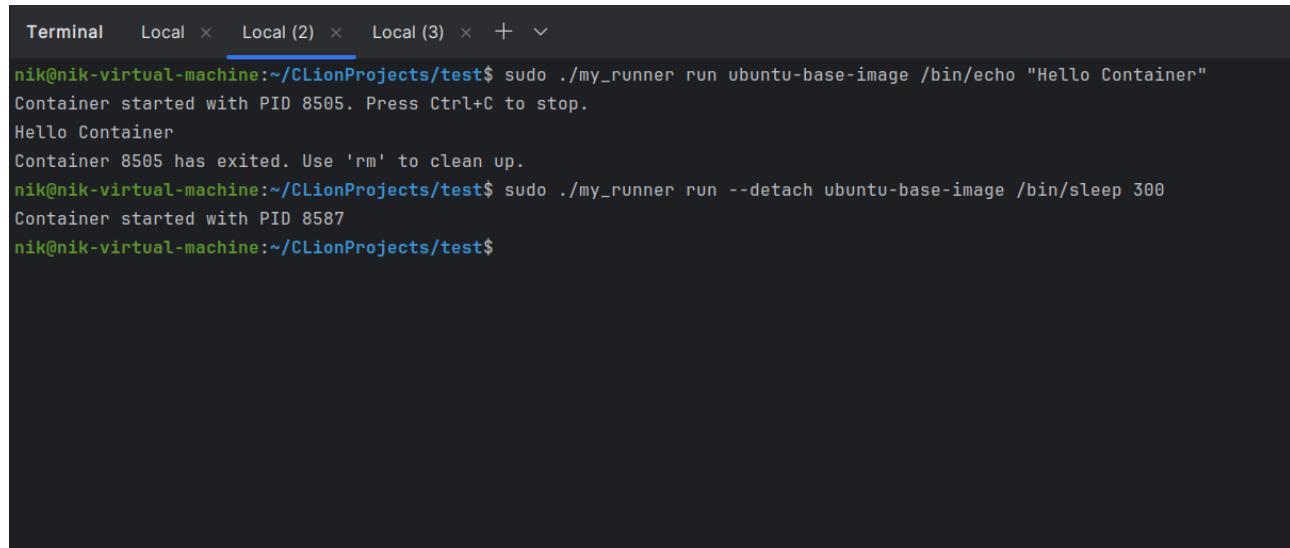
```
Terminal Local × Local (2) × Local (3) × + ~
nik@nik-virtual-machine:~/CLionProjects/test$ rm -f ebpf_log.txt
nik@nik-virtual-machine:~/CLionProjects/test$ sudo python3 monitor.py
Starting eBPF monitoring... Press Ctrl+C to exit.
```

شکل ۵۷: اجرای فایل مانیتور

مرحله دوم: ایجاد یک محفظه در حالی که مانیتور در حال اجراست، در ترمینال دوم دو محفظه‌ی ساده اجرا می‌کنیم. این عمل باعث تولید فراخوانی‌های سیستمی مورد نظر ما می‌شود.

```
sudo ./my_runner run ubuntu-base-image /bin/echo "Hello Container"
sudo ./my_runner run --detach ubuntu-base-image /bin/sleep 300
```

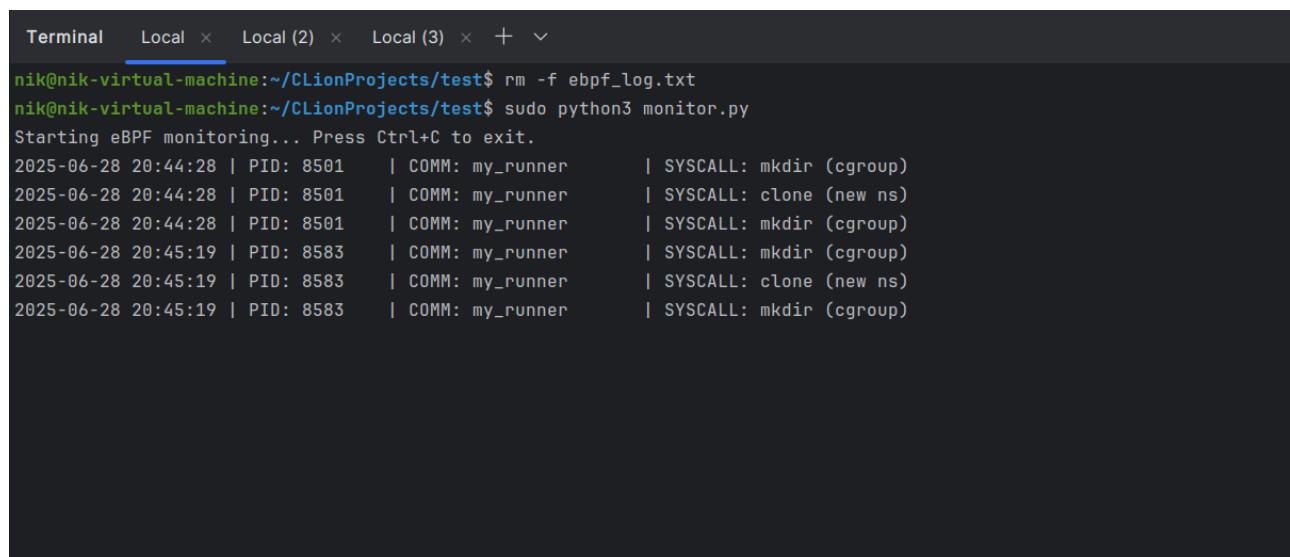
نتیجه: محفظه‌ها با موفقیت اجرا می‌شوند. همزمان، رویدادهای رهگیری‌شده به صورت زنده نمایش داده می‌شوند.



The screenshot shows a terminal window with three tabs: Local, Local (2), and Local (3). The Local (3) tab is active and displays the following command and its output:

```
nik@nik-virtual-machine:~/CLionProjects/test$ sudo ./my_runner run ubuntu-base-image /bin/echo "Hello Container"
Container started with PID 8505. Press Ctrl+C to stop.
Hello Container
Container 8505 has exited. Use 'rm' to clean up.
nik@nik-virtual-machine:~/CLionProjects/test$ sudo ./my_runner run --detach ubuntu-base-image /bin/sleep 300
Container started with PID 8587
nik@nik-virtual-machine:~/CLionProjects/test$
```

شکل ۵۸: اجرای محفظه‌ها



The screenshot shows a terminal window with three tabs: Local, Local (2), and Local (3). The Local (3) tab is active and displays the following command and its output:

```
nik@nik-virtual-machine:~/CLionProjects/test$ rm -f ebpf_log.txt
nik@nik-virtual-machine:~/CLionProjects/test$ sudo python3 monitor.py
Starting eBPF monitoring... Press Ctrl+C to exit.
2025-06-28 20:44:28 | PID: 8501 | COMM: my_runner | SYSCALL: mkdir (cgroup)
2025-06-28 20:44:28 | PID: 8501 | COMM: my_runner | SYSCALL: clone (new ns)
2025-06-28 20:44:28 | PID: 8501 | COMM: my_runner | SYSCALL: mkdir (cgroup)
2025-06-28 20:45:19 | PID: 8583 | COMM: my_runner | SYSCALL: mkdir (cgroup)
2025-06-28 20:45:19 | PID: 8583 | COMM: my_runner | SYSCALL: clone (new ns)
2025-06-28 20:45:19 | PID: 8583 | COMM: my_runner | SYSCALL: mkdir (cgroup)
```

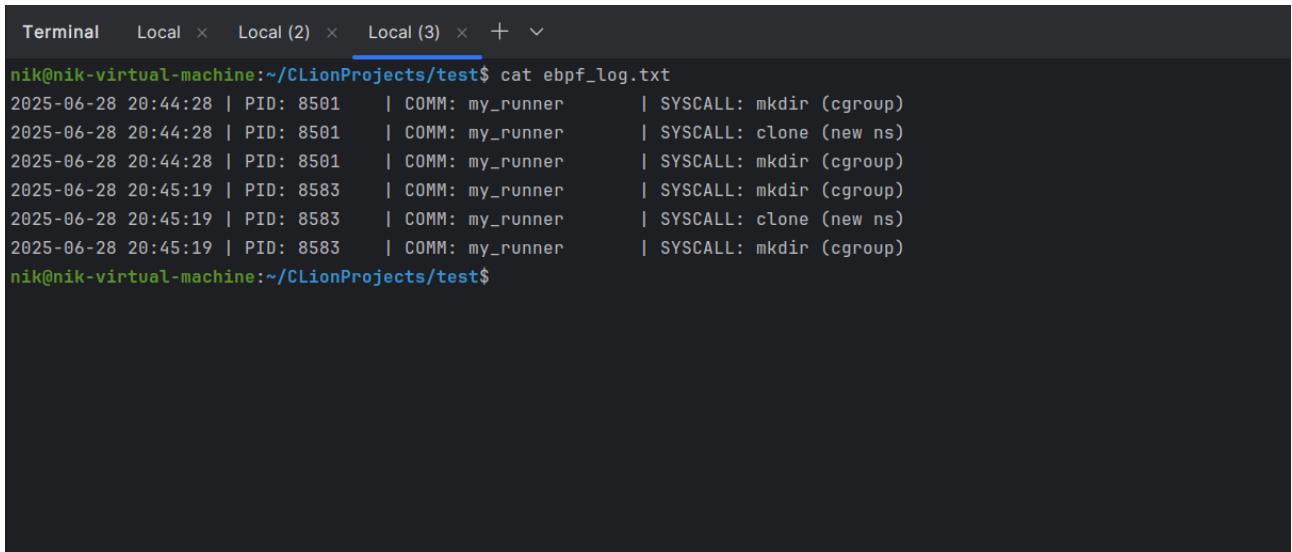
شکل ۵۹: مشاهده‌ی بی‌درنگ رویدادهای رهگیری‌شده توسط eBPF

مرحله سوم: بررسی فایل لاغ نهایی

پس از اتمام کار محفظه، اسکریپت مانیتور را با **Ctrl+C** متوقف کرده و محتويات فایل `ebpf_log.txt` را بررسی می‌کنیم.

```
cat ebpf_log.txt
```

نتیجه: فایل لاغ باید تنها شامل رویدادهای مرتبط با پروسه‌ی `my_runner` باشد. وجود حداقل یک رویداد `(new clone ns)` و یک رویداد `mkdir (cgroup)` در این فایل، صحبت عملکرد کامل سیستم نظارتی را تأیید می‌کند. فیلتر کردن رویدادها در سطح هسته باعث شده است که لاغ نهایی تمیز و بدون اطلاعات اضافی از سایر پروسه‌های سیستم باشد.



```
Terminal Local × Local (2) × Local (3) × + ▾
nik@nik-virtual-machine:~/CLionProjects/test$ cat ebpf_log.txt
2025-06-28 20:44:28 | PID: 8501 | COMM: my_runner | SYSCALL: mkdir (cgroup)
2025-06-28 20:44:28 | PID: 8501 | COMM: my_runner | SYSCALL: clone (new ns)
2025-06-28 20:44:28 | PID: 8501 | COMM: my_runner | SYSCALL: mkdir (cgroup)
2025-06-28 20:45:19 | PID: 8583 | COMM: my_runner | SYSCALL: mkdir (cgroup)
2025-06-28 20:45:19 | PID: 8583 | COMM: my_runner | SYSCALL: clone (new ns)
2025-06-28 20:45:19 | PID: 8583 | COMM: my_runner | SYSCALL: mkdir (cgroup)
nik@nik-virtual-machine:~/CLionProjects/test$
```

شکل ۶۰: فایل لاغ نهایی حاوی رویدادهای فیلترشده‌ی مربوط به `my_runner`

نتیجه‌گیری آزمایش این آزمایش نشان داد که ابزار نظارتی مبتنی بر *eBPF* به درستی پیاده‌سازی شده است. این سیستم قادر است با اتصال به نقاط رهگیری هسته، فرآخوانی‌های سیستمی مورد نظر را به صورت کارآمد فیلتر و رهگیری کند. این قابلیت، یک ابزار قدرتمند برای اشکال‌زدایی، تحلیل عملکرد و نظارت امنیتی بر روی خود سیستم مدیریت محفظه فراهم می‌آورد.

۴.۳ نقد و بررسی سیستم و پیشنهادهای جهت بهبود

در این بخش، به تحلیل چالش‌ها، محدودیت‌های فعلی سیستم و ارائه‌ی پیشنهادهایی برای توسعه و بهبود آن در آینده پرداخته می‌شود.

وابستگی به نسخه **cgroups** و محدودیت‌های آن سیستم فعلی به طور کامل بر پایه‌ی رابط مدرن و یکپارچه‌ی **v2 cgroups** طراحی شده است. این انتخاب، مدیریت منابع را به دلیل ساختار سلسه‌مراتبی واحد، ساده‌تر و منسجم‌تر می‌کند. با این حال، این طراحی یک وابستگی اساسی به سیستم میزبان ایجاد می‌کند و برنامه با سیستم‌هایی که هنوز از **cgroups v1** استفاده می‌کنند، سازگار نیست. علاوه بر این، با وجود برتری‌های فراوان، برخی از کنترلرهای موجود در نسخه‌ی اول، مانند کنترل پهنهای باند شبکه (**net_cls**) را به صورت مستقیم ارائه نمی‌دهد.

برای بهبود، می‌توان در آینده یک مکانیزم جایگزین برای کنترل ترافیک شبکه پیاده‌سازی کرد. یک راهکار استاندارد، ترکیب **eBPF** با ابزار کنترل ترافیک لینوکس (**tc**) است. در این روش، یک برنامه‌ی **eBPF** می‌تواند بسته‌های شبکه‌ی خروجی از یک **cgroup** خاص را شناسایی کرده و آن‌ها را با یک علامت مشخص، نشانه‌گذاری (**tag**) کند. سپس، با استفاده از یک صفت زمان‌بندی (**qdisc**) و فیلترهای (**tc**)، می‌توان برای بسته‌های نشانه‌گذاری شده، قوانین محدودسازی پهنهای باند را اعمال کرد. این رویکرد، انعطاف‌پذیری بسیار بالایی را برای مدیریت دقیق ترافیک شبکه فراهم می‌آورد.

مدیریت ایمیج و سادگی بیش از حد فایل سیستم پایه در حال حاضر، ایمیج پایه‌ی سیستم (**ubuntu-base-image**) یک دایرکتوری ساده است که با اسکریپت **setup_rootfs.sh** و با کمی کردن دستی چند باینری و کتابخانه‌های وابسته آن‌ها ساخته می‌شود. این رویکرد برای تست و نمایش مفاهیم پایه بسیار مفید است، اما برای کاربردهای واقعی بسیار محدود‌کننده است. سیستم قادر هرگونه مکانیزم مدیریت ایمیج، مانند لایه‌بندی، نسخه‌بندی، دانلود از رجیستری، یا ساخت ایمیج از روی یک فایل تعريف (مانند **Dockerfile**) است.

برای توسعه‌ی این بخش، می‌توان قابلیت پشتیبانی از استاندارد ایمیج (**OCI**) (*Open Container Initiative*) را به سیستم اضافه کرد. این کار به برنامه اجازه می‌دهد تا ایمیج‌های استاندارد را که به صورت فایل‌های **tar** بسته‌بندی شده‌اند، وارد (**import**) کند. در مراحل پیش‌رفته‌تر، می‌توان دستورات جدیدی مانند **pull** برای دانلود ایمیج‌ها از رجیستری‌های عمومی و **build** برای ساخت ایمیج‌های جدید بر اساس یک فایل تعريف، به رابط خط فرمان اضافه کرد.

شبکه‌بندی ایزووله و عدم ارتباط با دنیای خارج سیستم فعلی با ایجاد یک **Network Namespace** کاملاً ایزووله، امنیت بالایی را فراهم می‌کند، اما این ایزووله‌سازی کامل، مانع از هرگونه ارتباط محفظه با سیستم میزبان، سایر محفظه‌ها، یا شبکه‌ی خارجی می‌شود. تنها رابط فعل، رابط محلی (**lo**) است که برای ارتباطات داخلی خود محفوظه کاربرد دارد. یک راهکار استاندارد برای حل این مشکل، پیاده‌سازی یک شبکه‌ی مجازی با استفاده از یک جفت (**virtual Ethernet**) یا **veth** است. در این روش، یک سر این جفت در داخل **namespace** محفوظه قرار گرفته، یک آدرس **IP** خصوصی (مثلًا از رنج ۱۰/۰/۰/۲۴) به آن اختصاص داده می‌شود. سر دیگر این جفت روی سیستم میزبان باقی مانده و به یک پل معجزی لینوکسی (**Linux bridge**) متصل می‌گردد. تمام محفوظه‌ها به همین پل متصل می‌شوند و می‌توانند با یکدیگر ارتباط برقرار کنند. برای ارتباط با شبکه‌ی خارجی، باید قابلیت مسیریابی (**IP forwarding**) روی هاست فعل شده و قوانین ترجمه‌ی آدرس شبکه (**NAT**) با استفاده از **iptables** تنظیم شود تا ترافیک خروجی از محفوظه‌ها با آدرس **IP** هاست به دنیای خارج ارسال گردد.

مدیریت وضعیت شکننده و غیرپایدار وضعیت محفوظه‌ها (شامل پیکربندی و شناسه‌ی لایه‌ی **OverlayFS**) در مسیر **/run/my_runtime** ذخیره می‌شود. از آنجایی که محتوای دایرکتوری **/run** معمولاً از نوع **tmpfs** است، با راهاندازی مجدد سیستم میزبان، تمام این اطلاعات از بین می‌رود. این بدان معناست که محفوظه‌های متوقف شده پس از ریبوت قابل بازیابی نیستند. علاوه بر این، استفاده از **PID** به عنوان شناسه‌ی اصلی محفوظه، یک نقطه‌ی ضعف محسوب می‌شود، زیرا **PID**‌ها پس از خاتمه‌ی یک پروسه، توسط سیستم عامل بازیافت شده و ممکن است به پروسه‌ی دیگری تخصیص داده شوند.

برای بهبود، باید از یک مکانیزم ذخیره‌سازی پایدار استفاده کرد. مسیر استاندارد برای این کار، **/var/lib/my_runtime** است. مهم‌تر از آن، باید از یک شناسه‌ی منحصر به فرد و پایدار (**UUID**) برای هر محفوظه استفاده کرد. در این مدل، **PID** تنها وضعیت لحظه‌ای پروسه‌ی در حال اجرای یک محفوظه خواهد بود، در حالی که هویت اصلی محفوظه توسط **UUID** آن تعريف می‌شود.

عدم وجود ذخیره‌سازی پایدار (**Volumes**) در سیستم فعلی، تمام داده‌هایی که یک برنامه در داخل محفوظه ایجاد می‌کند، در لایه‌ی قابل نوشت **OverlayFS** آن ذخیره می‌شوند. این لایه به صورت ذاتی موقتی و گذرا است و با اجرای دستور **rm**، تمام این داده‌ها نیز از بین می‌روند. این معماری برای اجرای برنامه‌های بدون وضعیت (**stateless**) مناسب است، اما برای برنامه‌هایی که نیاز به ذخیره‌سازی پایدار دارند (مانند پایگاه‌های داده)، کاملاً نامناسب است.

برای رفع این محدودیت، باید قابلیت حجم‌های داده (**volumes**) به سیستم اضافه شود. این کار می‌تواند با افزودن یک فلگ جدید مانند **/path/on/host** – **/volume /path/on/host** – به دستور **run** پیاده‌سازی شود. در پشت صحنه، این قابلیت با استفاده از یک **mount** فراخوانی (**bind**) از نوع (**bind**) پیاده‌سازی می‌شود. این فراخوانی، یک دایرکتوری از سیستم میزبان را در یک مسیر مشخص در داخل فایل سیستم ادغام‌شده محفوظه مانت می‌کند. این کار به محفوظه اجازه می‌دهد تا داده‌های خود را مستقیماً روی دیسک هاست ذخیره کند، به طوری که حتی پس از حذف محفوظه نیز داده‌ها پایدار باقی بمانند.

۵.۳ نتیجہ گیری

در این پروژه، یک سیستم مدیریت محفظه‌ی کامل با معماری بدون دیمون (*daemonless*) از پایه و با استفاده از زبان برنامه‌نویسی C طراحی و پیاده‌سازی شد. هدف اصلی، ساخت ابزاری مشابه *Podman* بود که با تعامل مستقیم با قابلیت‌های سطح پایین هسته‌ی لینوکس، یک محیط ایزوله، امن و قابل مدیریت برای اجرای برنامه‌ها فراهم آورد. این پروژه به موفقیت به اهداف خود دست یافت و نشان داد که چگونه می‌توان با ترکیب فناوری‌های کلیدی لینوکس، یک راهکار کانتینری‌سازی مدرن و کارآمد ایجاد کرد.

در طول این پژوهش، تمام جنبه‌های بنیادی کانتینر سازی با موفقیت پیاده‌سازی و آزمایش شدند. با استفاده از فراخوانی سیستمی (`clone`)، فضاهای نام (`namespaces`) مختلفی شامل `IPC`، `Network`، `Mount`، `UTS`، `PID` و `User` برای ایجاد ایزووله سازی جامع به کار گرفته شدند. چالش‌های پیچیده‌هایی مانند `race condition` در هنگام راه اندازی `User Namespace`، با استفاده از مکانیزم همگام سازی (`pipe`) با موفقیت مدیریت شد. سیستم فایل هر محفظه با استفاده از `chroot` در یک زندان امن قوار گرفت و با بهره‌گیری از `OverlayFS`، یک معماری ایمیج لایه‌ای و کارآمد، مشابه تصاویر داکر، محقق گردید.

علاوه بر ایزو ول هسازی، مدیریت منابع نیز به صورت کامل با استفاده از رابط مدرن `cgroups v2` پیاده سازی شد. قابلیت محدود سازی حافظه، توان پردازنده و پهنهای باند و رویدی / خروجی به صورت دقیق و قابل انکا عمل کرد. بر روی این زیر ساخت، یک رابط خط فرمان `CLI` جامع با دستوراتی مانند `run`, `start`, `status`, `list`, `stop` و `rm` طراحی شد که چرخهی حیات کامل یک محفظه را به صورت کاربر پسند مدیریت می کند.

در نهایت، قابلیت‌های پیشرفته‌ای نیز به سیستم اضافه گردید. امکان استراکچری فضای نام *IPC*، انتشار رویدادهای *mount*، و قفل کردن پروسه روى یک هسته‌ی *CPU* خاص، انعطاف‌پذیری قابل توجهی را برای سناریوهای خاص فراهم آورد. همچنین، با پایه‌سازی یک ابزار نظارتی مبتنی بر *BPF*، نشان داده شد که چگونه می‌توان با سریار بسیار کم، فعالیت‌های سطح هسته‌ی مربوط به ایجاد محفظه‌ها را رهگیری و تحلیل کرد.

- https://ebpf.io/what_is_ebpf/
- <https://www.tigera.io/learn/guides/ebpf/>
- <https://www.datadoghq.com/knowledge-center/ebpf/>
- <https://www.infoq.com/articles/gentle-linux-ebpf-introduction/>
- <https://en.wikipedia.org/wiki/EBPF>
- <https://man7.org/linux/man-pages/man7/namespaces.7.html>
- https://en.wikipedia.org/wiki/Linux_namespaces
- <https://www.redhat.com/en/blog/7-linux-namespaces>
- <https://theboreddev.com/understanding-linux-namespaces/>
- <https://blog.quarkslab.com/digging-into-linux-namespaces-part-1.html>
- <https://how.dev/answers/what-are-kernel-namespaces>
- <https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces>
- <https://en.wikipedia.org/wiki/Cgroups>
- <https://man7.org/linux/man-pages/man7/cgroups.7.html>
- <https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html>
- <https://medium.com/%40weidagang/linux-beyond-the-basics-cgroups-f157d93bd755>
- <https://en.wikipedia.org/wiki/Chroot>
- <https://www.geeksforgeeks.org/chroot-command-in-linux-with-examples/>
- <https://www.cyberciti.biz/faq/unix-linux-chroot-command-examples-usage-syntax/>
- <https://www.makeuseof.com/chroot-in-linux/>
- <https://medium.com/%40weidagang/linux-beyond-the-basics-chroot-f831b20c93d7>
- <https://www.linuxfordevices.com/tutorials/linux/chroot-command-in-linux>
- <https://en.wikipedia.org/wiki/OverlayFS>
- <https://en.wikipedia.org/wiki/UnionFS>
- <https://news.ycombinator.com/item?id=21569168>
- <https://tunnelix.com/overlay-union-filesystems-in-linux/>
- <https://docs.kernel.org/filesystems/overlayfs.html>
- https://youtu.be/e17768BNUPw?si=Zx76K0-NPAxYE_Yn
- https://youtu.be/q_bQeXhWxZM?si=qalXh07aiHt4fSDY
- <https://youtu.be/eyNBf1sqdBQ?si=2JSCnqPXVA19Ki0s>