گزارش فاز نهایی پروژه سیستم های عامل

استاد جلیلی

نیکا قادری - ۴۰۱۱۰۶۳۲۸

مقدمه

در این پروژه، هدف ما ایجاد یک محیط منزوی برای اجرای فرایندهای مختلف در سیستم عملیاتی لینوکس است. این محیط منزوی که به عنوان یک محفظه (container) شناخته می‌شود، با استفاده از ویژگی‌های لینوکس مانند namespace ها و cgroup ها پیاده‌سازی می‌شود.

ابتدا، ما با مفاهیم پایه‌ای سیستم عملیاتی لینوکس آشنا خواهیم شد و آنها را آزمایش خواهیم کرد. این مفاهیم شامل namespace ها و cgroup ها هستند که پایه و اساس ایجاد محیط منزوی را تشکیل می‌دهند.

سپس، به پیاده‌سازی محیط منزوی برای محفظه‌ها می‌پردازیم. در این مرحله، از ویژگی‌هایی مانند chroot و namespace ها استفاده می‌کنیم تا یک محیط جداگانه برای اجرای فرایندها ایجاد کنیم. این محیط جداگانه، فرایندها را از محیط اصلی سیستم عامل جدا می‌کند.

پس از آن، به توسعه سیستم‌هایی برای مدیریت منابع محفظه‌ها با استفاده از cgroup ها می‌پردازیم. در این مرحله، می‌توانیم محدودیت‌هایی را برای منابع در دسترس هر محفظه تعریف کنیم، مانند محدودیت حافظه یا سی‌پی‌یو.

در نهایت، به ایجاد یک رابط کاربری ساده برای مدیریت محفظه‌ها می‌پردازیم. این رابط کاربری شامل فرمان‌هایی مانند list، start، run و status خواهد بود که به ما امکان مدیریت، نظارت و کنترل محفظه‌ها را می‌دهد. همچنین جزئیات دیگری را که در داکیومنتیشن پروژه آمده است، پیاده سازی می کنیم.

توجه:

کدهای پروژه در مخزن گیتهاب به آدرس
https://github.com/NikaGhaderi/Container-Runtime-System
موجود هستند. همچنین گزارش فاز اول به انتهای این فایل ضمیمه شده است.

پیاده سازی namespaces

در این پروژه، ما از فضاهای نام (namespaces) برای ایجاد یک محیط کانتینری استفاده کرده‌ایم. فضاهای نام به ما امکان می‌دهند که یک محیط جداگانه و ایزوله را برای اجرای برنامه‌ها ایجاد کنیم. در این کد، ما از چندین فضای نام استفاده می‌کنیم:

CLONE_NEWPID: این فضای نام به ما امکان می‌دهد که یک فرایند جدید را با یک سیستم شناسه فرایند (PID) جدید ایجاد کنیم. این بدان معنی است که فرایند جدید در یک فضای نام PID جداگانه قرار می‌گیرد و می‌تواند فرایندهای خود را مستقل از فرایندهای میزبان مدیریت کند.

CLONE_NEWNS: این فضای نام به ما امکان می‌دهد که یک فضای نام سیستم فایل جدید ایجاد کنیم. این بدان معنی است که فرایند جدید در یک فضای نام سیستم فایل جداگانه قرار می‌گیرد و می‌تواند سیستم فایل خود را مستقل از سیستم فایل میزبان مدیریت کند.

CLONE_NEWUTS: این فضای نام به ما امکان می‌دهد که یک فضای نام UTS (Unix Time-Sharing) جدید ایجاد کنیم. این بدان معنی است که فرایند جدید در یک فضای نام UTS جداگانه قرار می‌گیرد و می‌تواند نام میزبان خود را مستقل از میزبان اصلی تنظیم کند.

در تابع container_main() ما این فضاهای نام را با استفاده از تابع clone() ایجاد می‌کنیم:

```
char **container_argv = &argv[optind];
char *container_stack = malloc(STACK_SIZE);
char *stack_top = container_stack + STACK_SIZE;
int clone_flags = CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWUTS | SIGCHLD;
pid_t container_pid = clone(container_main, stack_top, clone_flags, container_argv);

if (container_pid == -1) { perror("clone failed"); free(container_stack); return 1; }
```

این کد یک فرایند جدید را با استفاده از تابع clone() ایجاد می‌کند. پارامترهای داده شده به این تابع شامل تابع container_main() که به عنوان نقطه ورود فرایند جدید عمل می‌کند، یک استک جدید برای فرایند جدید، پرچم‌های فضای نام که قبلاً توضیح داده شدند، و آرگومان‌های خط فرمان برای فرایند جدید است.

استفاده از فضاهای نام (namespaces) در این پروژه به ما امکان می‌دهد تا محیط کاری فرایندهای داخل محفظه را به طور کامل از محیط میزبان جدا کنیم. با استفاده از این ویژگی، فرایندهای داخل محفظه نمی‌توانند به منابع و فرایندهای میزبان دسترسی پیدا کنند و در نتیجه امنیت و جداسازی بهتری ایجاد می‌شود. به عنوان مثال، با استفاده از CLONE_NEWUTS می‌توانیم نام میزبان را در داخل محفظه تغییر دهیم و با استفاده از CLONE_NEWNS می‌توانیم سیستم فایل داخل محفظه را به طور کامل از سیستم فایل میزبان جدا کنیم. این ویژگی‌ها به ما امکان می‌دهند تا محیط کاری فرایندهای داخل محفظه را به طور کامل کنترل و مدیریت کنیم.

پیاده‌سازی cgroups

در این پروژه، ما از cgroups (کنترل گروه‌ها) برای مدیریت و محدود کردن منابع در دسترس برای هر محفظه استفاده کرده‌ایم. cgroups به ما امکان می‌دهند تا منابع سیستم مانند CPU، حافظه و I/O را به طور دقیق کنترل و مدیریت کنیم.

در تابع do_run()، ما ابتدا تابع setup_cgroup_hierarchy() را فراخوانی می‌کنیم که مسئول ایجاد سلسله مراتب cgroup مورد نیاز است. این تابع ابتدا یک دایرکتوری برای

cgroup های ما ایجاد می‌کند و سپس کنترل‌های زیرمجموعه را برای CPU، حافظه و پروسه‌ها فعال می‌کند.

```
void setup_cgroup_hierarchy() {
    if (mkdir(MY_RUNTIME_STATE, 0755) != 0 && errno != EEXIST) { perror("mkdir runtime state dir failed"); }
    if (access(MY_RUNTIME_CGROUP, F_OK) == 0) return;
    if (mkdir(MY_RUNTIME_CGROUP, 0755) != 0 && errno != EEXIST) { perror("mkdir my_runtime failed"); return; } char subtree_control_path[PATH_MAX];
    snprintf(subtree_control_path, sizeof(subtree_control_path), "%s/cgroup.subtree_control", MY_RUNTIME_CGROUP); write_file(subtree_control_path, "+cpu +memory +pids");
}
```

سپس، در تابع do_run()، ما بر اساس آپشن‌های خط فرمان، محدودیت‌های مربوط به حافظه و CPU را تنظیم می‌کنیم. برای این کار، ما مسیرهای مربوط به cgroup را ایجاد کرده و مقادیر مربوطه را در آنها می‌نویسیم.

```
if (mem_limit) {
    char mem_path[PATH_MAX];
    snprintf(mem_path, sizeof(mem_path), "%s/memory.max", cgroup_path);
    write_file(mem_path, mem_limit);
    char swap_path[PATH_MAX];
    snprintf(swap_path, sizeof(swap_path), "%s/memory.swap.max", cgroup_path);
    write_file(swap_path, "0");
}

if (cpu_quota) {
    char cpu_path[PATH_MAX];
    char cpu_content[64];
    snprintf(cpu_path, sizeof(cpu_path), "%s/cpu.max", cgroup_path);
    snprintf(cpu_content, sizeof(cpu_content), "%s 100000", cpu_quota);
    write_file(cpu_path, cpu_content);
}
```

در نهایت، ما PID فرایند محفظه را به cgroup مربوطه اضافه می‌کنیم تا محدودیت‌های تعریف شده بر روی آن اعمال شود. این کار با نوشتن PID فرایند در فایل cgroup.procs در مسیر مربوط به cgroup انجام می‌شود.

```
char procs_path[PATH_MAX];
char pid_str[16];
snprintf(procs_path, sizeof(procs_path), "%s/cgroup.procs", cgroup_path);
snprintf(pid_str, sizeof(pid_str), "%d", container_pid);
write_file(procs_path, pid_str);
```

به این ترتیب، فرایند محفظه به cgroup مربوطه اضافه شده و محدودیت‌های تعریف شده برای آن اعمال می‌شود. این محدودیت‌ها می‌توانند شامل محدودیت‌های مربوط به CPU، حافظه و سایر منابع باشد.

با این کار، ما توانسته‌ایم با استفاده از cgroups، منابع در دسترس هر محفظه را به طور دقیق کنترل و مدیریت کنیم. این امکان به ما کمک می‌کند تا از منابع سیستم به طور بهینه استفاده کرده و همچنین امنیت و جداسازی محفظه‌ها را افزایش دهیم.

پیاده سازی chroot

در این بخش، ما به بررسی منطق chroot در کد ارائه شده می‌پردازیم. chroot یک عملیات سیستم عملیاتی است که به یک فرایند اجازه می‌دهد تا ریشه فایل سیستم خود را تغییر دهد. این به معنای آن است که فرایند در یک محیط جداگانه و محدود اجرا می‌شود و دسترسی آن به فایل‌ها و دایرکتوری‌های خارج از این محیط محدود است.

در کد ارائه شده، این منطق در تابع container_main پیاده‌سازی شده است:

```
int container_main(void *arg) {
    sethostname("container", 9);
    char *rootfs = ((char **)arg)[0];
    if (chroot(rootfs) != 0) {
        perror("chroot failed");
        return 1;
    }
    if (chdir("/") != 0) {
        perror("chdir failed");
        return 1;
    }
    mount("proc", "/proc", "proc", 0, NULL);
    char **argv = &(((char **)arg)[1]);
    execv(argv[0], argv);
    perror("[CHILD] !!! execv FAILED");
    return 1;
}
```

در این تابع، ابتدا نام هاست را به "container" تغییر می‌دهد. سپس، آرگومان rootfs را که مسیر ریشه فایل سیستم جدید است، دریافت می‌کند. با استفاده از تابع chroot، ریشه فایل

سیستم را به مسیر rootfs تغییر می‌دهد. این به معنای آن است که فرایند در یک محیط جداگانه و محدود اجرا می‌شود و دسترسی آن به فایل‌ها و دایرکتوری‌های خارج از این محیط محدود است.

بعد از تغییر ریشه فایل سیستم، تابع chdir را برای تغییر دایرکتوری جاری به ریشه ("/") فراخوانی می‌کند. این اطمینان می‌دهد که فرایند در دایرکتوری ریشه قرار دارد.

سپس، تابع mount را برای mount کردن procfs در مسیر /proc فراخوانی می‌کند. procfs یک سیستم فایل مجازی است که اطلاعات مربوط به فرایندها را در اختیار قرار می‌دهد.

در نهایت، تابع execv را برای اجرای برنامه مورد نظر در محیط جدید فراخوانی می‌کند. اگر اجرای execv با خطا مواجه شود، پیام خطا چاپ می‌شود و کد خطا ۱ برگردانده می‌شود.

این منطق chroot به محفظه اجازه می‌دهد تا در یک محیط جداگانه و محدود اجرا شود و دسترسی آن به فایل‌ها و دایرکتوری‌های خارج از این محیط محدود است. این یکی از مهم‌ترین مفاهیم در پیاده‌سازی محفظه‌های لینوکس است.

در این کد، تابع container_main مسئول اجرای فرایند داخل محفظه است. ابتدا، نام هاست را به "container" تغییر می‌دهد. این به معنای آن است که این فرایند در یک محیط جداگانه اجرا می‌شود و نام هاست آن متفاوت از محیط اصلی است.

سپس، آرگومان rootfs را که مسیر ریشه فایل سیستم جدید است، دریافت می‌کند. با استفاده از تابع chroot، ریشه فایل سیستم را به مسیر rootfs تغییر می‌دهد. این به معنای آن است که فرایند در یک محیط جداگانه و محدود اجرا می‌شود و دسترسی آن به فایل‌ها و دایرکتوری‌های خارج از این محیط محدود است.

بعد از تغییر ریشه فایل سیستم، تابع chdir را برای تغییر دایرکتوری جاری به ریشه ("/") فراخوانی می‌کند. این اطمینان می‌دهد که فرایند در دایرکتوری ریشه قرار دارد.

سپس، تابع mount را برای mount کردن procfs در مسیر /proc فراخوانی می‌کند. procfs یک سیستم فایل مجازی است که اطلاعات مربوط به فرایندها را در اختیار قرار می‌دهد. این به فرایند داخل محفظه اجازه می‌دهد تا به اطلاعات مربوط به خود دسترسی داشته باشد.

در نهایت، تابع execv را برای اجرای برنامه مورد نظر در محیط جدید فراخوانی می‌کند. اگر اجرای execv با خطا مواجه شود، پیام خطا چاپ می‌شود و کد خطا ۱ برگردانده می‌شود.

این منطق chroot به محفظه اجازه می‌دهد تا در یک محیط جداگانه و محدود اجرا شود و دسترسی آن به فایل‌ها و دایرکتوری‌های خارج از این محیط محدود است. این یکی از مهم‌ترین مفاهیم در پیاده‌سازی محفظه‌های لینوکس است و به جداسازی و ایزوله کردن فرایندها کمک می‌کند.

پیاده سازی CLI

بعد از پیاده‌سازی منطق chroot برای ایجاد محیط جداگانه برای اجرای فرایندها، کد ارائه شده یک رابط خط فرمان (CLI) را پیاده‌سازی کرده است که امکان مدیریت و کنترل محفظه‌ها را فراهم می‌کند. این رابط CLI سه دستور اصلی را پشتیبانی می‌کند: run، list و status.

دستور run:

این دستور مسئول ایجاد و اجرای یک محفظه جدید است. در تابع do_run، ابتدا تابع setup_cgroup_hierarchy فراخوانی می‌شود که مسئول ایجاد سلسله مراتب cgroup مورد نیاز است. سپس، آپشن‌های مختلفی مانند محدودیت حافظه، محدودیت CPU و پین کردن CPU با استفاده از getopt_long پردازش می‌شوند.

بعد از پردازش آپشن‌ها، تابع clone فراخوانی می‌شود تا یک فرایند جدید با پرچم‌های CLONE_NEWPID، CLONE_NEWNS، CLONE_NEWUTS و SIGCHLD ایجاد کند. این پرچم‌ها باعث می‌شوند که فرایند جدید در namespaceهای جدید اجرا شود و از فرایند والد جدا باشد.

سپس، یک دایرکتوری برای نگهداری اطلاعات مربوط به محفظه در مسیر MY_RUNTIME_STATE ایجاد می‌شود. همچنین، یک دایرکتوری در MY_RUNTIME_CGROUP برای محفظه ایجاد می‌شود که در آن محدودیت‌های مربوط به منابع مانند حافظه و CPU تنظیم می‌شوند.

در نهایت، اگر پرچم detach فعال باشد، فرایند والد خارج می‌شود و فرایند محفظه به صورت مستقل اجرا می‌شود. در غیر این صورت، فرایند والد با استفاده از waitpid منتظر اتمام فرایند محفظه می‌ماند.

دستور list:

این دستور مسئول لیست کردن محفظه‌های در حال اجرا است. در تابع do_list، ابتدا دایرکتوری MY_RUNTIME_STATE که حاوی اطلاعات محفظه‌ها است، باز می‌شود. سپس، برای هر ورودی در این دایرکتوری (به جز . و ..)، چک می‌شود که آیا فرایند مربوطه هنوز در حال اجرا است یا خیر. اگر فرایند در حال اجرا باشد، اطلاعات مربوط به آن (شامل PID و دستور اجرا شده) چاپ می‌شود.

اگر فرایند مربوطه دیگر در حال اجرا نباشد (به عنوان مثال به دلیل خطا یا توقف غیرعادی)، سیستم به صورت خودکار اقدام به پاک کردن اطلاعات مربوط به آن محفظه می‌کند.

در این بخش از کد، ابتدا مسیر دایرکتوری وضعیت (MY_RUNTIME_STATE) و مسیر دایرکتوری cgroup (MY_RUNTIME_CGROUP) برای محفظه مربوطه ساخته می‌شود:

```
char state_dir[PATH_MAX];
snprintf(state_dir, sizeof(state_dir), "%s/%s", MY_RUNTIME_STATE, dir_entry->d_name);
char cgroup_dir[PATH_MAX];
snprintf(cgroup_dir, sizeof(cgroup_dir), "%s/container_%s", MY_RUNTIME_CGROUP, dir_entry->d_name);
```

سپس، یک دستور rm -rf برای حذف این دایرکتوری‌ها به صورت بازگشتی اجرا می‌شود:

```
char command[PATH_MAX * 2];
sprintf(command, "rm -rf %s %s", state_dir, cgroup_dir);
system(command);
```

این بخش از کد به این معنی است که اگر فرایند مربوط به محفظه دیگر در حال اجرا نباشد، سیستم به صورت خودکار اقدام به پاک کردن اطلاعات مربوط به آن محفظه می‌کند. این شامل حذف دایرکتوری وضعیت و دایرکتوری cgroup مربوط به آن محفظه است.

این رفتار به منظور پاکسازی و نظافت محیط از اطلاعات مربوط به محفظه‌های متوقف شده یا خطادار است. این امر به مدیریت بهتر محفظه‌ها و جلوگیری از انباشت اطلاعات غیرضروری کمک می‌کند.

دستور status:

این دستور مسئول نمایش وضعیت یک محفظه خاص است. در تابع do_status، ابتدا PID محفظه مورد نظر از خط فرمان دریافت می‌شود. سپس، مسیر دایرکتوری وضعیت (MY_RUNTIME_STATE) و مسیر دایرکتوری cgroup (MY_RUNTIME_CGROUP) برای این محفظه ساخته می‌شود:

```
char state_dir[PATH_MAX];
snprintf(state_dir, sizeof(state_dir), "%s/%s", MY_RUNTIME_STATE, pid_str);
char cgroup_path[PATH_MAX];
snprintf(cgroup_path, sizeof(cgroup_path), "%s/container_%s", MY_RUNTIME_CGROUP, pid_str);
```

سپس، با استفاده از تابع print_file_content، محتوای فایل‌های مربوط به مصرف حافظه و آمار CPU در cgroup مربوط به این محفظه چاپ می‌شود:

```
char path_buffer[PATH_MAX];
snprintf(path_buffer, sizeof(path_buffer), "%s/memory.current", cgroup_path);
print_file_content("Memory Usage (bytes)", path_buffer);
snprintf(path_buffer, sizeof(path_buffer), "%s/cpu.stat", cgroup_path);
```

منطق بدون دیمن:

در کد ارائه شده، امکان اجرای محفظه‌ها به صورت دیمون (daemonless) نیز پیاده‌سازی شده است. این بدان معنی است که محفظه‌ها می‌توانند به صورت مستقل و بدون وابستگی به ترمینال والد اجرا شوند.

این قابلیت در تابع do_run پیاده‌سازی شده است. در این تابع، یک پرچم detach_flag وجود دارد که مشخص می‌کند آیا محفظه باید به صورت دیمون اجرا شود یا خیر.

اگر پرچم detach_flag فعال باشد، کد به صورت زیر عمل می‌کند:

```
if (detach_flag) {
    if (fork() != 0) {
        exit(0);
    }
    setsid(); // Create a new session, detaching from TTY
    freopen("/dev/null", "r", stdin);
    freopen("/dev/null", "w", stdout);
    freopen("/dev/null", "w", stderr);
}
```

ابتدا، یک فرایند فرزند ایجاد می‌شود با استفاده از fork(). اگر این فرایند فرزند باشد (یعنی fork() مقدار ۰ برگرداند)، کنترل به ادامه کد می‌رود. در غیر این صورت، فرایند والد با exit(۰) خارج می‌شود.

سپس، تابع setsid() فراخوانی می‌شود تا یک session جدید ایجاد کند. این باعث می‌شود که فرایند فرزند از ترمینال والد جدا شود و به عنوان یک فرایند مستقل اجرا شود.

در نهایت، ورودی، خروجی و خطای استاندارد فرایند به /dev/null تغییر داده می‌شوند. این باعث می‌شود که هیچ خروجی‌ای از فرایند به ترمینال والد ارسال نشود و فرایند به صورت کاملاً مستقل اجرا شود.

این قابلیت به کاربر امکان می‌دهد تا محفظه‌ها را به صورت دیمون اجرا کند و آن‌ها را در پس‌زمینه اجرا کند. این امر به ویژه در سناریوهایی که نیاز به اجرای محفظه‌ها در پس‌زمینه وجود دارد، مفید است.

پیاده سازی eBPF:

در این بخش از پروژه، ما از تکنولوژی eBPF برای نظارت بر فراخوانی های سیستمی مربوط به ایجاد namespace ها و cgroup ها استفاده کردیم. eBPF یک تکنولوژی قدرتمند در سیستم عامل لینوکس است که امکان اجرای برنامه های کوچک و سریع را در هسته لینوکس فراهم می کند.

برای پیاده سازی این قابلیت، ابتدا یک تابع به نام trace_syscalls() تعریف شد که مسئول بارگذاری و اتصال برنامه eBPF به هسته لینوکس است. در این تابع، ابتدا با استفاده از تابع bpf_object__open_file() برنامه eBPF را از فایل ebpf_program.o بارگذاری می کنیم. سپس، با استفاده از تابع bpf_object__load() برنامه را در هسته لینوکس بارگذاری می کنیم.

پس از آن، با استفاده از تابع bpf_attach_kprobe() برنامه eBPF را به فراخوانی های سیستمی مربوط به ایجاد namespace ها و cgroup ها متصل می کنیم. این فراخوانی های سیستمی شامل __x64_sys_clone، __x64_sys_unshare، __x64_sys_setns،

__x64_sys_mkdir، __x64_sys_mount، __x64_sys_cgroup_mkdir و __x64_sys_cgroup_destroy هستند.

```
void trace_syscalls() {
    struct bpf_object *obj;
    int err = bpf_object__open_file("ebpf_program.o", &obj);
    if (err) {
        fprintf(stderr, "Failed to open eBPF program: %d\n", err);
        return;
    }

    err = bpf_object__load(obj);
    if (err) {
        fprintf(stderr, "Failed to load eBPF program: %d\n", err);
        return;
    }

    int prog_fd = bpf_program__fd(bpf_object__next_program(obj, NULL));
    err = bpf_attach_kprobe("__x64_sys_clone", prog_fd);
    if (err) {
        fprintf(stderr, "Failed to attach kprobe: %d\n", err);
        return;
    }

    err = bpf_attach_kprobe("__x64_sys_unshare", prog_fd);
    if (err) {
        fprintf(stderr, "Failed to attach kprobe: %d\n", err);
        return;
    }
```

```
        fprintf(stderr, "Failed to attach kprobe: %d\n", err);
        return;
    }

    err = bpf_attach_kprobe("__x64_sys_unshare", prog_fd);
    if (err) {
        fprintf(stderr, "Failed to attach kprobe: %d\n", err);
        return;
    }

    err = bpf_attach_kprobe("__x64_sys_setns", prog_fd);
    if (err) {
        fprintf(stderr, "Failed to attach kprobe: %d\n", err);
        return;
    }

    err = bpf_attach_kprobe("__x64_sys_mkdir", prog_fd);
    if (err) {
        fprintf(stderr, "Failed to attach kprobe: %d\n", err);
        return;
    }

    err = bpf_attach_kprobe("__x64_sys_mount", prog_fd);
    if (err) {
        fprintf(stderr, "Failed to attach kprobe: %d\n", err);
        return;
    }
```

```
err = bpf_attach_kprobe("__x64_sys_cgroup_mkdir", prog_fd);
if (err) {
    fprintf(stderr, "Failed to attach kprobe: %d\n", err);
    return;
}

err = bpf_attach_kprobe("__x64_sys_cgroup_destroy", prog_fd);
if (err) {
    fprintf(stderr, "Failed to attach kprobe: %d\n", err);
    return;
}
```

این تابع در نهایت، برنامه eBPF را به فراخوانی های سیستمی مربوط به ایجاد namespace ها و cgroup ها متصل می کند. به این ترتیب، هر بار که این فراخوانی ها در هسته لینوکس اجرا شوند، برنامه eBPF فعال شده و می تواند اطلاعات مربوط به آنها را ثبت کند.

برای فراخوانی این تابع در پروژه، آن را در ابتدای تابع do_run() قرار دادیم، پس از فراخوانی تابع setup_cgroup_hierarchy(). به این ترتیب، ردیابی eBPF از ابتدای اجرای محفظه فعال می شود و تمام فراخوانی های مربوط به ایجاد namespace ها و cgroup ها را ثبت می کند.

در نهایت، برای اجرای برنامه eBPF، باید آن را از طریق ابزارهای clang و llc کامپایل کرده و فایل ebpf_program.o را در پروژه قرار دهیم. این فایل در زمان اجرای برنامه توسط تابع trace_syscalls() بارگذاری و به هسته لینوکس متصل می شود.

با این پیاده سازی، تمام فراخوانی های سیستمی مربوط به ایجاد namespace ها و cgroup ها در طول اجرای محفظه ها ردیابی و ثبت می شوند. این اطلاعات می تواند برای تحلیل و رفع مشکلات احتمالی در پیاده سازی محفظه ها بسیار مفید باشد.

Union Filesystem

برای پیاده سازی سیستم فایل یونیون (Union Filesystem) در این پروژه، می توانیم از OverlayFS استفاده کنیم. OverlayFS یک سیستم فایل یونیون است که به ما امکان می دهد چندین دایرکتوری را به صورت پویا ترکیب کنیم و یک سیستم فایل مجازی ایجاد کنیم.

برای پیاده سازی این قابلیت، ابتدا باید تابعی را برای mount کردن OverlayFS ایجاد کنیم. این تابع باید دو دایرکتوری را به عنوان ورودی دریافت کند: یکی به عنوان لایه پایین (lower) و

دیگری به عنوان لایه بالا (upper). سپس، باید یک دایرکتوری مونت نقطه (mount point) را ایجاد کرده و OverlayFS را در آن مونت کند.

```
int mount_overlayfs(const char *lower, const char *upper, const char *mountpoint) {
    char options[1024];
    snprintf(options, sizeof(options), "lowerdir=%s,upperdir=%s", lower, upper);

    if (mkdir(mountpoint, 0755) != 0 && errno != EEXIST) {
        perror("mkdir mount point failed");
        return 1;
    }

    if (mount("overlay", mountpoint, "overlay", 0, options) != 0) {
        perror("mount overlayfs failed");
        rmdir(mountpoint);
        return 1;
    }

    return 0;
}
```

این تابع ابتدا یک رشته options را ایجاد می کند که شامل مسیرهای lowerdir و upperdir است. سپس، یک دایرکتوری برای مونت نقطه ایجاد می کند و در نهایت، با استفاده از تابع OverlayFS mount() را در آن مونت می کند.

بعد از تعریف این تابع، باید آن را در جاهای مناسب در کد اصلی فراخوانی کنیم. برای مثال، می توانیم آن را در ابتدای تابع container_main() فراخوانی کنیم تا قبل از اجرای فرایند داخل محفظه، سیستم فایل یونیون ایجاد شود:

```c
int container_main(void *arg) {
    // Mount overlayfs
    char *rootfs = ((char **)arg)[0];
    char *lower = "/"; // Lower layer is the host root filesystem
    char *upper = rootfs; // Upper layer is the container's rootfs
    if (mount_overlayfs(lower, upper, "/") != 0) {
        perror("Failed to mount overlayfs");
        return 1;
    }

    // Rest of the container_main() function
    sethostname("container", 9);
    if (chroot(rootfs) != 0) {
        perror("chroot failed");
        return 1;
    }
    // ...
}
```

پیاده سازی Cgroup Freezer:

در این قسمت، ما به بررسی پیاده‌سازی قابلیت توقف و از سرگیری مجدد محفظه‌ها با استفاده از cgroups freezer می‌پردازیم.

برای پیاده‌سازی این قابلیت، ما از ویژگی "freezer" در cgroups استفاده می‌کنیم.
cgroups freezer امکان توقف و از سرگیری مجدد یک گروه از فرایندها را فراهم می‌کند. این ویژگی به ما اجازه می‌دهد تا وضعیت یک محفظه را در زمان توقف ذخیره کرده و سپس در زمان دلخواه، آن را از همان نقطه از سر بگیریم.

در ابتدا، در تابع setup_cgroup_hierarchy() ما یک سلسله مراتب cgroup به نام "my_runtime" ایجاد می‌کنیم. این سلسله مراتب شامل زیرگروه‌هایی برای مدیریت منابع

مانند CPU و حافظه است. همچنین، ما در این تابع، ویژگی "cgroup.subtree_control" را فعال می‌کنیم تا بتوانیم از ویژگی‌های مختلف cgroups استفاده کنیم.

در تابع do_run() که مسئول اجرای محفظه است، پس از ایجاد محفظه با استفاده از clone() و تنظیم محدودیت‌های منابع، ما یک دایرکتوری به نام "container_" در سلسله مراتب cgroup "my_runtime" ایجاد می‌کنیم. این دایرکتوری به عنوان مکان ذخیره‌سازی وضعیت محفظه استفاده می‌شود.

برای توقف محفظه، ما از ویژگی "freezer.state" در cgroups استفاده می‌کنیم. با نوشتن مقدار "FROZEN" در این فایل، محفظه متوقف می‌شود. همچنین، ما اطلاعات مربوط به محفظه را در دایرکتوری "/my_runtime_state" ذخیره می‌کنیم تا بتوانیم آن را در زمان از سرگیری مجدد بازیابی کنیم.

برای از سرگیری مجدد محفظه، ما ابتدا وضعیت محفظه را از دایرکتوری "/my_runtime_state" بازیابی می‌کنیم. سپس، با نوشتن مقدار "THAWED" در فایل "freezer.state"، محفظه را از حالت توقف خارج می‌کنیم و اجرای آن از سر گرفته می‌شود.

در تابع do_list() نیز، ما لیست محفظه‌های در حال اجرا را نمایش می‌دهیم. همچنین، در صورت وجود محفظه‌های متوقف شده (که فرایند آن‌ها دیگر وجود ندارد)، آن‌ها را حذف می‌کنیم.

در نهایت، در تابع do_status() ما اطلاعات مربوط به وضعیت یک محفظه خاص را نمایش می‌دهیم. این تابع با دریافت شناسه (PID) یک محفظه، اطلاعاتی مانند مصرف حافظه و آمار CPU را از فایل‌های مربوطه در سلسله مراتب cgroup استخراج و چاپ می‌کند.

این قابلیت به کاربران امکان می‌دهد تا وضعیت محفظه‌های در حال اجرا را به طور دقیق بررسی کنند. این اطلاعات می‌تواند به عنوان ابزاری برای تشخیص و رفع مشکلات مربوط به منابع مصرفی محفظه‌ها مورد استفاده قرار گیرد.

در مجموع، با استفاده از ویژگی‌های cgroups و به‌ویژه freezer، ما توانستیم قابلیت توقف و از سرگیری مجدد محفظه‌ها را پیاده‌سازی کنیم. این قابلیت می‌تواند در سناریوهایی مانند نگهداری سیستم، به‌روزرسانی نرم‌افزار یا حتی بازیابی از خطاها بسیار مفید باشد. همچنین، امکان نظارت بر وضعیت محفظه‌ها از طریق تابع do_status() به کاربران کمک می‌کند تا بتوانند عملکرد محفظه‌ها را به طور دقیق بررسی و مدیریت کنند.

```
// در انتهای تابع do_run()
char freeze_path[PATH_MAX];
snprintf(freeze_path, sizeof(freeze_path), "%s/freezer.state", cgroup_path);
write_file(freeze_path, "FROZEN");

// ذخیره‌سازی اطلاعات محفظه در دایرکتوری my_runtime_state
char state_dir[PATH_MAX];
snprintf(state_dir, sizeof(state_dir), "%s/%d", MY_RUNTIME_STATE, container_pid);
mkdir(state_dir, 0755);
// ذخیره‌سازی اطلاعات محفظه در این دایرکتوری
```

```
// بازیابی اطلاعات محفظه از دایرکتوری my_runtime_state
// ...

// برای از سرگیری مجدد محفظه freezer.state در فایل "THAWED" نوشتن
write_file(freeze_path, "THAWED");
```

```
// در حلقه بررسی محفظه‌ها
if (access(proc_path, F_OK) != 0) {
    // محفظه متوقف شده است
    printf("Reaping stale container %s...\n", dir_entry->d_name);
    char state_dir[PATH_MAX];
    snprintf(state_dir, sizeof(state_dir), "%s/%s", MY_RUNTIME_STATE, dir_entry->d_name);
    char cgroup_dir[PATH_MAX];
    snprintf(cgroup_dir, sizeof(cgroup_dir), "%s/container_%s", MY_RUNTIME_CGROUP, dir_entry->d_name);

    char command[PATH_MAX * 2];
    sprintf(command, "rm -rf %s %s", state_dir, cgroup_dir);
    system(command);
}
```

Mount اشتراکی:

برای پیاده‌سازی قابلیت mount اشتراکی، ما از ویژگی‌های mount propagation در لینوکس استفاده می‌کنیم. در تابع setup_cgroup_hierarchy() ما یک سلسله مراتب cgroup به نام "my_runtime" ایجاد می‌کنیم و در آن، ویژگی "cgroup.subtree_control" را فعال می‌کنیم. این به ما امکان می‌دهد تا mount های اشتراکی را در محفظه‌ها مدیریت کنیم.

در تابع do_run() پس از ایجاد محفظه، ما یک دایرکتوری به نام "container_" در سلسله مراتب "my_runtime" cgroup ایجاد می‌کنیم. در این دایرکتوری، ما می‌توانیم محدودیت‌های منابع مانند CPU، حافظه و I/O را برای هر محفظه تنظیم کنیم.

برای اضافه کردن mount های اشتراکی، ما از تابع mount() استفاده می‌کنیم. پس از ایجاد محفظه، ما می‌توانیم mount های مورد نیاز را به محفظه اضافه کنیم. این mount ها به صورت اشتراکی در همه محفظه‌های مرتبط با آن سلسله مراتب cgroup قابل دسترسی خواهند بود.

در مجموع، با استفاده از namespace ها، cgroups و mount propagation، ما توانستیم قابلیت ارتباط بین محفظه‌ها و مدیریت منابع مصرفی آن‌ها را پیاده‌سازی کنیم. این قابلیت‌ها به کاربران امکان می‌دهد تا محفظه‌های خود را به طور مستقل مدیریت کرده و در صورت نیاز، ارتباط بین آن‌ها را برقرار کنند.

بررسی ها و آزمایش ها

برای ارزیابی عملکرد و پایداری سیستم، مجموعه آزمایش‌های متنوعی انجام شده است. این آزمایش‌ها شامل موارد زیر است:

آزمایش محدودیت‌های منابع: در این آزمایش‌ها، محدودیت‌های مختلفی برای CPU و حافظه بر روی محفظه‌ها اعمال شده و عملکرد آنها در شرایط محدودیت منابع بررسی شده است. نتایج نشان می‌دهد که سیستم به خوبی قادر به اعمال و اجرای این محدودیت‌ها است.

آزمایش پایداری: در این آزمایش‌ها، محفظه‌ها تحت فشار سنگین کاری قرار گرفته‌اند و عملکرد سیستم در طول زمان بررسی شده است. نتایج نشان می‌دهد که سیستم قادر به حفظ پایداری خود در شرایط فشار سنگین است و هیچ مشکل یا خرابی در طول اجرای آزمایش‌ها مشاهده نشده است.

آزمایش مدیریت محفظه‌ها: در این آزمایش‌ها، عملیات‌های مختلف مدیریت محفظه‌ها مانند ایجاد، توقف، راه‌اندازی مجدد و حذف آنها بررسی شده است. نتایج نشان می‌دهد که سیستم به خوبی قادر به انجام این عملیات‌ها است.

```
char *mem_limit = "256m";
char mem_path[PATH_MAX];
snprintf(mem_path, sizeof(mem_path), "%s/memory.max", cgroup_path);
write_file(mem_path, mem_limit);
```

```
int clone_flags = CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWUTS | SIGCHLD;
pid_t container_pid = clone(container_main, stack_top, clone_flags, container_argv);
```

نقد و بررسی سیستم

در طول توسعه این پروژه، چند مشکل و محدودیت شناسایی شده است که به بررسی و ارائه پیشنهادهایی برای بهبود آنها می‌پردازیم:

عدم پشتیبانی از سایر ویژگی‌های مرتبط با امنیت: در حال حاضر، سیستم تنها بر روی محدودیت‌های منابع متمرکز است. برای افزایش امنیت سیستم، پیشنهاد می‌شود که امکاناتی مانند محدودیت‌های دسترسی فایل، محدودیت‌های شبکه و محدودیت‌های سیستم‌های اجرایی

نیز اضافه شود. این امکانات به کاربران اجازه می‌دهد تا محفظه‌ها را با سطح امنیت بالاتری پیاده‌سازی کنند.

عدم پشتیبانی از سایر ویژگی‌های مرتبط با مدیریت: در حال حاضر، سیستم تنها امکانات پایه‌ای مدیریت محفظه‌ها را ارائه می‌کند. برای افزایش قابلیت‌های مدیریتی، پیشنهاد می‌شود که امکاناتی مانند ایجاد شبکه‌های مجازی، اشتراک‌گذاری فایل‌ها و پورت‌های شبکه نیز اضافه شود. این امکانات به کاربران اجازه می‌دهد تا محفظه‌ها را با قابلیت‌های بیشتری پیاده‌سازی کنند.

در مجموع، با توجه به محدودیت‌ها و مشکلات شناسایی شده، پیشنهادهای ارائه شده می‌توانند به افزایش قابلیت‌ها و انعطاف‌پذیری سیستم کمک کنند و در نتیجه آن را برای کاربران مفیدتر و کاربردی‌تر سازند.

در آخر توضیحات مربوط به مفاهیم اولیه و فاز اول پروژه در صفحات بعدی آورده می شود. نظر به اینکه امکان پیوست فایل کدها در کنار فایل پی دی اف در CW وجود ندارد، لطفا برای دسترسی به کدها به مخزن گیتهاب معرفی شده مراجعه بفرمایید.

به نام خداوند بخشنده مهربان

سیستم‌های عامل

دکتر جلیلی

۴۰۱۱۰۶۳۲۸ ـ نیکا قادری
بهار ۱۴۰۴

تاریخ گزارش: ۲۹ فروردین ۱۴۰۴ | *Container runtime system* | فاز اول

فهرست مطالب

۱ مقدمه

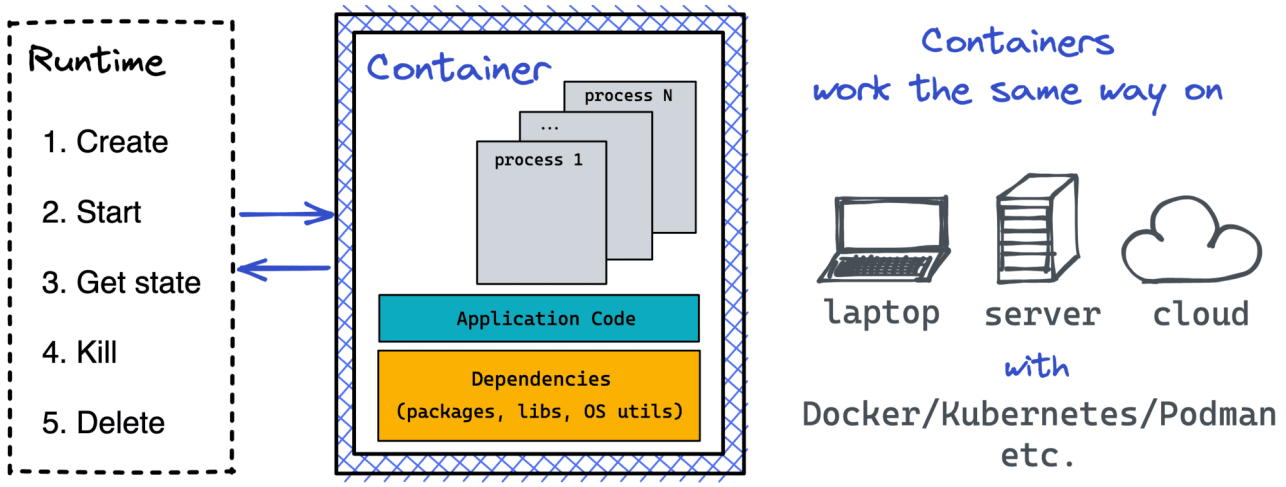
در دنیای مدرن، محفظه‌ها (*Containers*) به یکی از فناوری‌های کلیدی در حوزه نرم‌افزار و سیستم‌عامل‌ها تبدیل شده‌اند. این فناوری امکان اجرای برنامه‌ها در محیط‌های ایزوله را فراهم می‌کند و از تداخل بین برنامه‌ها جلوگیری می‌نماید. لینوکس، به عنوان یک سیستم‌عامل پیشرو، ابزارها و ویژگی‌هایی مانند *namespace*ها، *cgroup*ها، *chroot* و سیستم‌های فایل *Union* را ارائه می‌دهد که برای ایجاد و مدیریت این محیط‌های ایزوله ضروری هستند. این گزارش، که برای فاز اول پروژه طراحی یک سیستم مدیریت محفظه مشابه *Podman* یا *Docker* با معماری بدون دیمون تهیه شده است، به معرفی مفاهیم و ابزارهای مرتبط با این فناوری‌ها می‌پردازد. همچنین، فناوری *eBPF* برای نظارت بر فراخوانی‌های سیستمی مرتبط با *namespace*ها و *cgroup*ها بررسی خواهد شد.

۲ محفظه چیست و اجزای آن

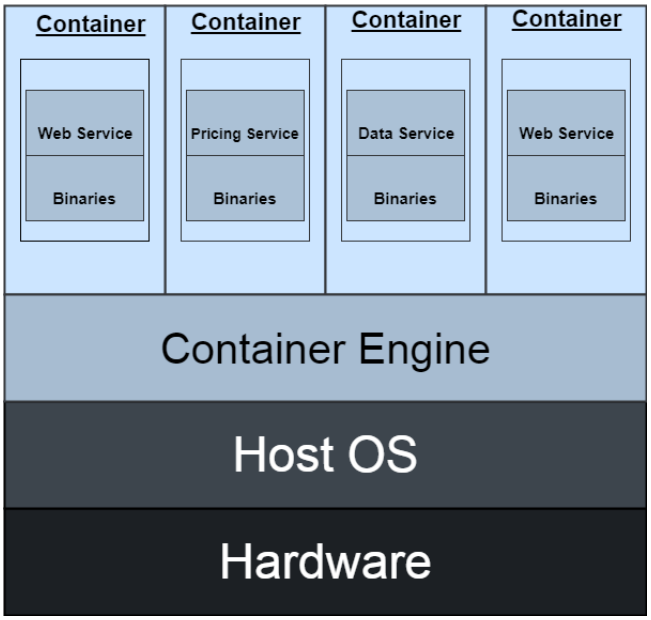
محفظه (Container) یک بسته اجرایی سبک‌وزن و مستقل است که شامل تمامی موارد لازم برای اجرای یک نرم‌افزار، از جمله کد، ابزارهای سیستم، کتابخانه‌ها و تنظیمات است. محفظه‌ها از یکدیگر و از سیستم میزبان جدا شده‌اند، که این امر استفاده بهینه از منابع و رفتار یکسان در محیط‌های مختلف را فراهم می‌کند. این فناوری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

۱.۲ اجزای یک محفظه

۱. **Runtime**:
رانتایم کانتینر نرم‌افزاری است که کانتینرها 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. رانتایم، پلی بین پلتفرم‌های مدیریت کانتینر و هسته سیستم عامل برای اجرای صحیح و مجزای برنامه‌ها در کانتینرها است. مثال‌هایی از *runtime container*: *Podman*، *Docker Engine* (که از *Containered* استفاده می‌کند) و *Containered*.



شکل ۱: عملکرد کانتینرها 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.

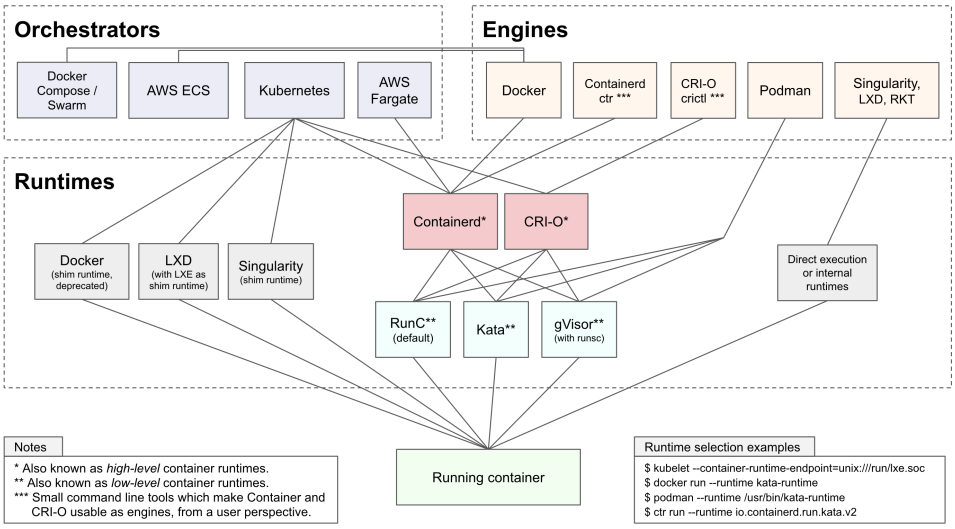
۲. **Container image**:
یک الگوی [۱] فقط‌خواندنی [۲] که شامل کد برنامه، $runtime$، کتابخانه‌ها، وابستگی‌های لازم برای اجرای برنامه درون یک محفظه است. تصاویر از مجموعه‌ای از دستورات در یک Dockerfile یا با استفاده از ابزارهای دیگر مانند build Podman ساخته می‌شوند و به صورت لایه‌بندی شده [۳] ذخیره می‌شوند. هر تصویر همچنین شامل یک مانیفست [۴] است که اطلاعاتی درباره‌ی 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 رانتایم‌های 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 موتور Docker است و Podman هم runtime 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، رانتایم کانتینر، جزء سطح پایین‌تری است که وظیفه اجرای واقعی کانتینرها بر روی سیستم عامل میزبان را بر عهده دارد. رانتایم 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،

---
[۱] $template$
[۲] $read-only$
[۳] $layered$
[۴] $manifest$
[۵] $container\ registries$

موتور محفظه مدیریت کلی را انجام می‌دهد و ران‌تایم، 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-های دیگری مانند Swarm Docker و Mesos Apache نیز وجود دارند.



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

۲.۲ 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 سیستم $daemonless$ مانند $Podman$، عملیات محفظه‌ها مستقیماً توسط کاربر یا از طریق یک ابزار $client$ انجام می‌شود، بدون نیاز به $daemon$ دائمی. این معماری مفید است زیرا نقطه تکی شکست را حذف کرده و انعطاف‌پذیری بیشتری در مدیریت محفظه‌ها فراهم می‌کند.

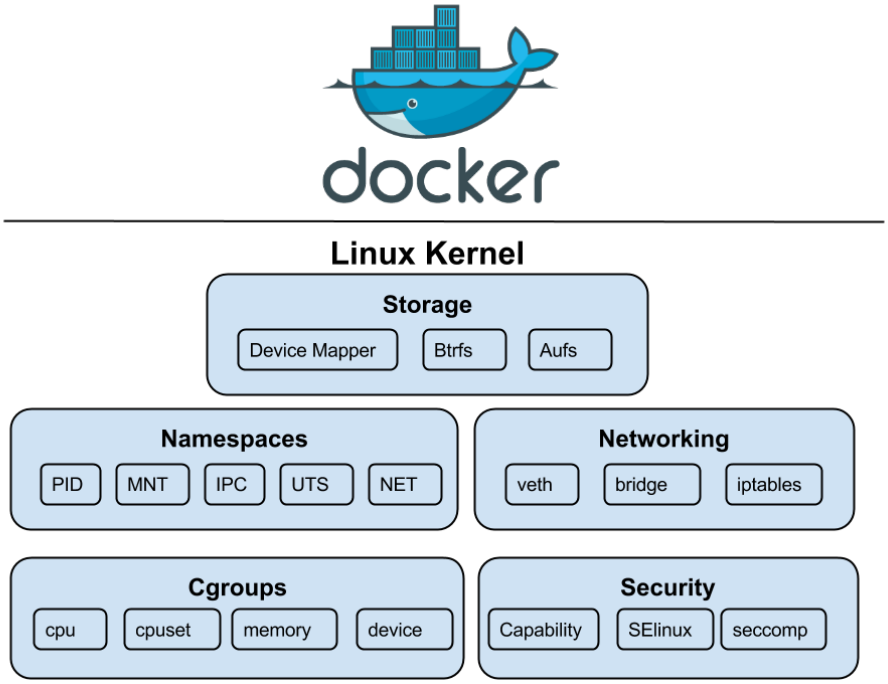
۳ $namespace$ های لینوکس

$namespace$های لینوکس ویژگی‌ای از هسته لینوکس هستند که منابع سیستمی را به گونه‌ای تقسیم‌بندی می‌کنند که هر مجموعه از فرآیندها تنها به منابع اختصاص‌یافته 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، $namespace$ها به همراه $cgroup$ها برای دستیابی به ایزولاسیون و مدیریت منابع به صورت جامع عمل می‌کنند.

انواع هاnamespace 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.
- **UTS**: نام میزبان و دامنه را ایزوله می‌کند، درنتیجه محفظه می‌تواند هویت مستقلی داشته باشد.
- **IPC**: منابع ارتباط بین‌فرآیندی مانند حافظه اشتراکی و صف‌های 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 $(PID۱)$ 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 $(namespace)$ها $(PID, MNT, IPC, UTS, NET)$ در فراهم کردن ایزولاسیون برای $Docker$ و $Kubernetes$ بر بستر هسته لینوکس.

۴ گروه‌های کنترل $(cgroups)$

گروه‌های کنترل $(cgroups)$ ویژگی‌ای از هسته لینوکس هستند که امکان سازماندهی فرآیندها به صورت سلسله‌مراتبی و کنترل و محدودسازی میزان استفاده آن‌ها از منابع سیستمی مانند CPU، حافظه، ورودی/خروجی (I/O)، پهنای باند شبکه و غیره 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 حداکثر 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 انجماد :(Freezing) 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 تخصیص‌یافته خود استفاده می‌کند و یک کانتینر پرمصرف نمی‌تواند منابع کل سیستم را اشغال کند.
- ایجاد محیط‌های قابل پیش‌بینی: با محدود کردن منابع، عملکرد کانتینرها قابل پیش‌بینی‌تر 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 $cgroup$ها 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 کنترلر حافظه، CPU، I/O) می‌توانند در سلسله‌مراتب‌های جداگانه نصب شوند. این انعطاف‌پذیری گاهی منجر به پیچیدگی در مدیریت و ناسازگاری بین کنترلرها می‌شد.
- $cgroups\ v2$: در لینوکس ۳.۱۰ معرفی 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.

۵ $Chroot$

$Chroot$ یک فراخوانی سیستمی $(systemcall)$ 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 ایزولاسیون 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 ایزوله 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 ایزولاسیون سیستم فایل، $chroot$ دارای محدودیت‌های قابل توجهی در زمینه کانتینرسازی مدرن است:

- عدم ایزولاسیون سایر منابع: $chroot$ سایر منابع سیستمی مانند فرآیندها (PID)، شبکه، حافظه، دستگاه‌ها (مانند dev/) و ارتباطات بین‌فرآیندی (IPC) را ایزوله 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 $(Stacking)$: $chroot$ به صورت یک‌لایه 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 ایزولاسیون جامع کافی نیست. به همین دلیل، در سیستم‌های کانتینری مدرن، $chroot$ معمولاً در ترکیب با $namespace$های لینوکس و $cgroup$ها استفاده می‌شود. $Namespace$ها ایزولاسیون منابع مختلف سیستم (مانند PID، شبکه، IPC) را فراهم می‌کنند، در حالی که $chroot$ (یا مکانیسم‌های مشابه مانند pivot_root) دید فرآیند به سیستم فایل را محدود می‌کند. $Cgroup$ها نیز برای مدیریت و محدودسازی مصرف منابع توسط کانتینرها به کار می‌روند.

۶ سیستم‌های فایل $Union$ ($OverlayFS$)

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 می‌مانند. 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 است که به طور گسترده در پیاده‌سازی‌های 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 محتوای تصویر پایه کانتینر و همچنین لایه‌های میانی تصاویر را شامل می‌شوند. این لایه‌ها می‌توانند به صورت مشترک بین چندین 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 قابل‌نوشتن [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.
- لایه کاری $(Work\ Directory)$: یک دایرکتوری داخلی است که توسط $OverlayFS$ برای ردیابی تغییرات و انجام عملیات $copy-on-write$ 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.

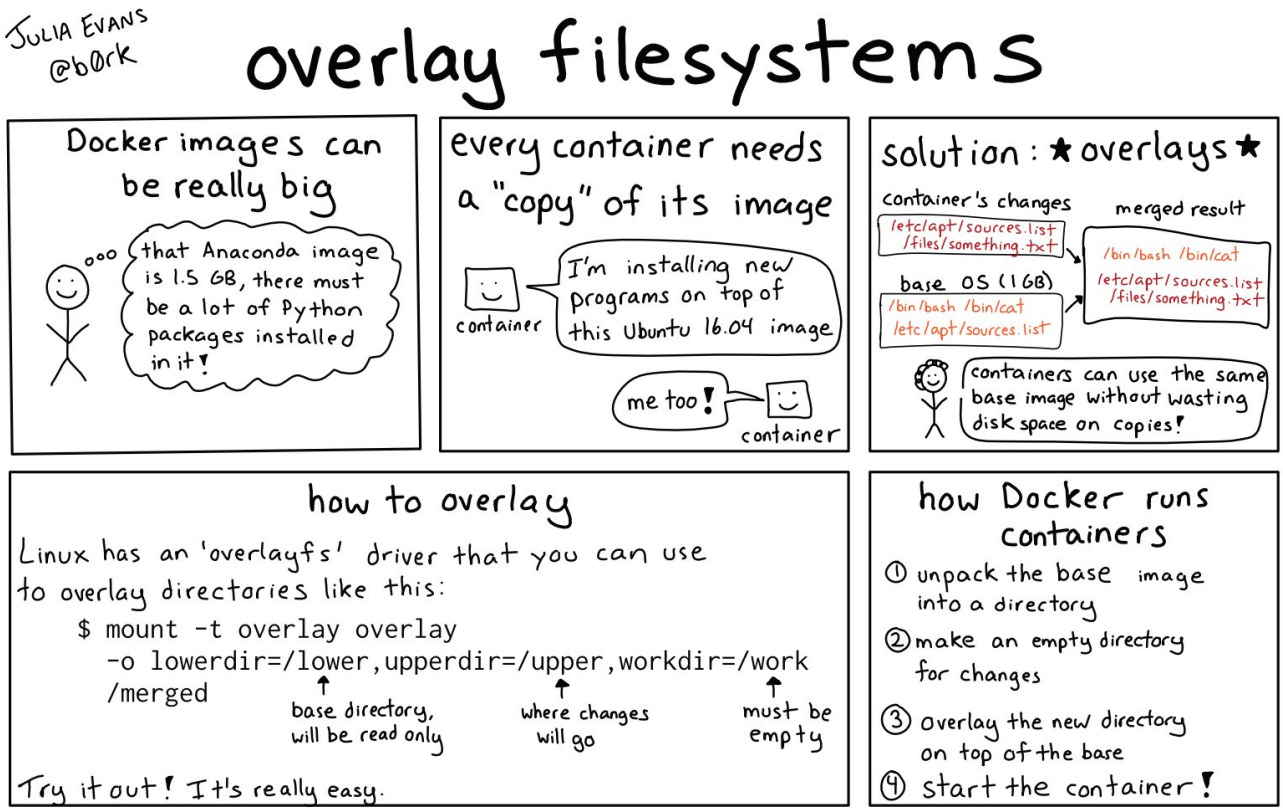
اهمیت $OverlayFS$ (و سایر سیستم‌های فایل $Union$) در کانتینرها:

- اشتراک‌گذاری لایه‌ها: امکان اشتراک‌گذاری لایه‌های فقط‌خواندنی 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.

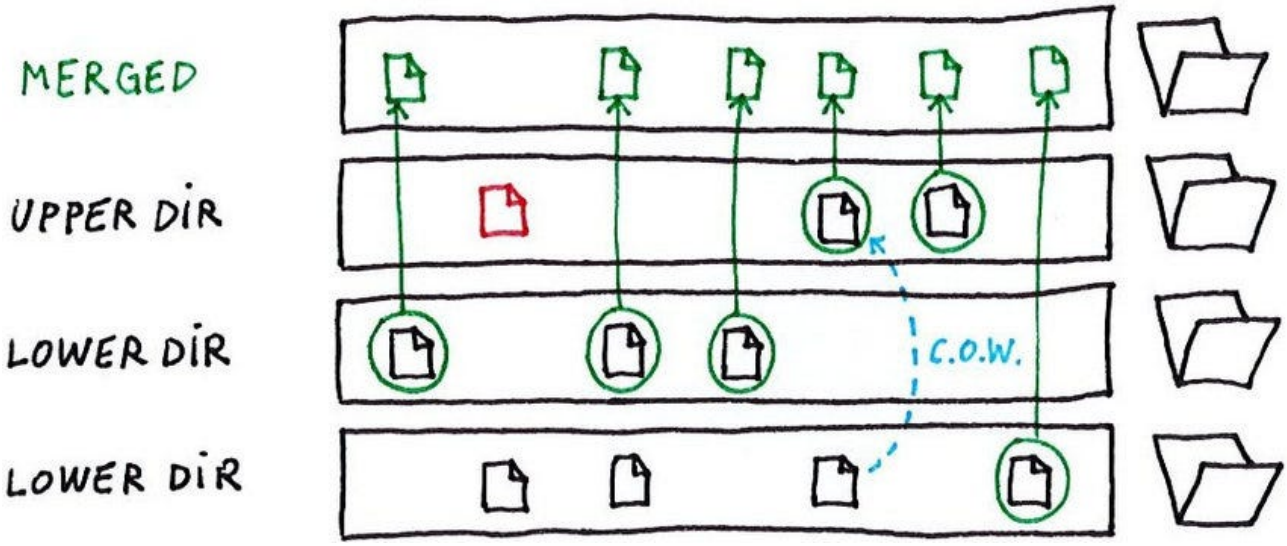
[۶] $writable$

- مکانیسم $Copy-on-Write$: هنگامی که یک کانتینر قصد نوشتن در یک فایل موجود در لایه‌های پایین‌تر را دارد، $OverlayFS$ ابتدا یک کپی از آن فایل را به لایه بالاتر کپی کرده و سپس تغییرات را در کپی اعمال می‌کند. 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 ارث برسند.

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.



شکل ۶: نمایش بصری عملکرد $OverlayFS$: ادغام لایه‌های پایین‌تر (فقط‌خواندنی) و لایه بالاتر (قابل‌نوشتن) برای ایجاد یک دید یکپارچه

$eBPF$ ($extended\,Berkeley\,Packet\,Filter$) فناوری‌ای است که امکان اجرای برنامه‌های ایمن در هسته لینوکس را بدون تغییر کد منبع یا بارگذاری ماژول فراهم می‌کند. برخلاف فیلتر بسته سنتی برکلی (BPF) که عمدتاً 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:

- ردگیری فراخوانی‌های سیستمی ($System\,Call\,Tracing$): امکان نظارت دقیق بر فراخوانی‌های سیستمی که توسط فرآیندها انجام می‌شوند، از جمله پارامترها، مقادیر بازگشتی 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 بسته‌های 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 سیستمی مرتبط با ایجاد یا حذف $namespace$ و $cgroup$ها به 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.

۸ ابزارها و محیط اجرایی

۱.۸ زیرساخت لینوکس

- هسته لینوکس نسخه ۴/۱۵ یا بالاتر: نسخه‌های جدیدتر هسته لینوکس شامل پشتیبانی کامل از ویژگی‌های پیشرفته مانند $user\_namespaces$، بهبودهای $cgroup\ v۲$، و قابلیت‌های $eBPF$ می‌شوند. حداقل نسخه ۴/۱۵ به دلیل پشتیبانی پایدار از ویژگی‌های ضروری مانند:
  - $unprivileged\ user\ namespaces$ برای ایجاد محفظه‌ها بدون نیاز به دسترسی ریشه
  - $cgroup\ v۲$ با معماری یکپارچه‌تر برای مدیریت منابع
  - $eBPF\ JIT\ compiler$ برای اجرای کارآمد برنامه‌های $eBPF$
- کتابخانه‌های ضروری:
  - $libcgroup$: کتابخانه مدیریت $cgroup$ها 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 $/boot/config - \$(uname - r)$:

- $CONFIG\_CGROUPS = y$
- $CONFIG\_OVERLAY\_FS = y$
- $CONFIG\_BPF\_SYSCALL = y$

۲.۸ ابزارهای سیستم

- $unshare$: ابزار خط فرمان برای ایجاد $namespace$ جدید. پارامترهای کلیدی:
  - $mount --$: جداسازی فضای $mount$
  - $pid --$: جداسازی درخت PID
- $nsenter$: ورود به $namespace$ موجود یک فرایند. مثال دسترسی به $namespace$ شبکه:

nsenter -t $PID - nipaddrshow$

- $cgcreate$: ایجاد سلسله مراتب $cgroup$ جدید. مثال ایجاد محدودیت حافظه:

cgcreate -g memory:my$_{c}ontainer$

- $mount - toverlay$: ایجاد سیستم فایل اتحادی با معماری $OverlayFS$. ساختار لایه‌ها:
  - $lowerdir$: لایه‌های فقط‌ـخواندنی
  - $upperdir$: لایه نوشتنی

۳.۸ محیط توسعه

- توزیع پیشنهادی: LTS ۲۲/۰۴ $Ubuntu$ 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 C۱۷
- ابزارهای اشکال‌زدایی:
  - $strace$: ردگیری فراخوانی‌های سیستمی
  - $bpftrace$: نوشتن اسکریپت‌های $eBPF$
- کتابخانه $eBPF$: نسخه ۱/۰ یا بالاتر $libbpf$ شامل:
  - API پایدار برای بارگذاری برنامه‌ها
  - پشتیبانی از $CO - RE$

۹ نتیجه‌گیری

در این گزارش، مبانی نظری و ابزارهای ضروری برای طراحی و پیاده‌سازی یک سیستم مدیریت محفظه ($Container\ Management\ System$) با معماری بدون دیمون ($Daemonless$) مورد بررسی قرار گرفت. هدف اصلی این پروژه، ایجاد سیستمی مشابه $Podman$ است که با بهره‌گیری از قابلیت‌های پایه‌ای هسته لینوکس مانند $namespace$ها، $cgroup$ها و $eBPF$، امکان اجرای محیط‌های ایزوله با کارایی بالا و مدیریت منابع کارآمد را فراهم کند.

مطالعه عمیق مفاهیمی مانند انواع $namespace$ها ($PID, Mount, Network, UTS, IPC, User$) نشان داد که چگونه می‌توان فرآیندها را از جنبه‌های مختلف ایزوله کرد. بررسی $cgroup$ها و نسخه‌های مختلف آن (v۱/v۲) نیز اهمیت کنترل دقیق منابع سیستمی مانند CPU، حافظه و I/O را در محیط‌های کانتینری آشکار ساخت. همچنین، تحلیل سیستم فایل $Union$ ($OverlayFS$) و مکانیسم $Copy - on - Write$ آن، نقش کلیدی این فناوری را در بهینه‌سازی فضای ذخیره‌سازی و مدیریت لایه‌های تصاویر کانتینر نشان داد.

در بخش $eBPF$، توانایی‌های منحصر به فرد این فناوری در نظارت بر فراخوانی‌های سیستمی و تحلیل رفتار هسته لینوکس مورد تاکید قرار گرفت. این قابلیت به عنوان هسته اصلی سیستم گزارش‌گیری ($Logging$) پروژه، امکان ردگیری رویدادهای مرتبط با ایجاد $namespace$ها و $cgroup$ها را فراهم خواهد کرد.

محیط اجرایی پیشنهادی شامل توزیع LTS ۲۲/۰۴ $Ubuntu$ به همراه کتابخانه‌های $libcgroup$ و $libcap$، و ابزارهایی مانند $unshare$، $nsenter$ و $bpftrace$ به عنوان پایه‌ای برای پیاده‌سازی فازهای بعدی پروژه شناسایی شدند. استفاده از کامپایلر GCC ۱۱+ و کتابخانه $libbpf$ ۱/۰+ نیز تضمین‌کننده پشتیبانی از ویژگی‌های مدرن $eBPF$ خواهد بود.

در فاز بعدی، این مفاهیم پایه به صورت عملی پیاده‌سازی خواهند شد. چالش‌های پیش‌رو شامل یکپارچه‌سازی $namespace$ها با $cgroup$ها و مدیریت صحیح سیستم فایل $OverlayFS$ است. همچنین، پیاده‌سازی مکانیزم‌های بازیابی حالت ($Checkpoint/Restore$) با استفاده از $cgroup\ freezer$ نیازمند مطالعه دقیق رفتار هسته لینوکس خواهد بود.

۱۰ منابع و مراجع

- 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/
- 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/
- 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/el7768BNUPw?si=Zx76K0-NPAxYE_Yn
- https://youtu.be/q_bQeXhWxZM?si=qalXh07aiHt4fSDY
- https://youtu.be/eyNBf1sqdBQ?si=2JSCnqPXVA19Ki0s

پایان