۱ مقدمه و اهداف پروژه

SimpleContainer یک سیستم اجرای کانتینر بدون دیمون (Daemonless-Container-Runtime) است که از ابتدا به زبان C پیاده‌سازی شده و با هدف ارائهٔ یک راه‌حل سبک، کارآمد و قابل‌فهم برای کانتینری‌سازی در لینوکس طراحی شده است. برخلاف سیستم‌هایی مانند Docker که به یک دیمون دائمی وابسته‌اند، Simple-Container مستقیماً از ویژگی‌های کرنل لینوکس بهره می‌برد.

اهداف اصلی پروژه:

- ایزوله‌سازی کامل با استفاده از namespaceهای لینوکس
- مدیریت منابع با cgroup v2
- ایزولاسیون فایل‌سیستم با chroot و overlayfs
- طراحی معماری بدون دیمون
- مانیتورینگ سطح پایین با eBPF
- رابط خط فرمان ساده برای کاربران

۲ معماری سیستم

SimpleContainer از چند مؤلفه‌ی مستقل تشکیل شده که از طریق هدرها با یکدیگر در ارتباط هستند:

- **container.c** : هستهٔ مدیریت کانتینرها
- **namespace.c** : تنظیم هایnamespace مختلف
- **cgroup.c** : مدیریت منابع با cgroup ۲v
- **filesystem.c** : پیاده‌سازی فایل‌سیستم جداگانه با overlayfs
- **cli.c** : رابط خط فرمان برای اجرا و کنترل
- **monitor.c** : نظارت با eBPF
- **ipc.c** : پشتیبانی از ارتباط بین‌پروسه‌ای
- **utils.c** : توابع کمکی عمومی

۳ نحوه پیاده‌سازی

۱.۳ namespace ها

namespace ها یکی از قابلیت‌های کلیدی کرنل لینوکس هستند که امکان جداسازی منابع بین کانتینرها را فراهم می‌کنند. در SimpleContainer از ۶ نوع namespace استفاده شده است:

۱. PID-Namespace : ایزوله‌سازی فضای شناسه‌های فرآیند که به هر کانتینر اجازه می‌دهد فهرست جداگانه‌ای از فرآیندها را داشته باشد. فرآیند اصلی کانتینر، PID ۱ خواهد بود:

```
// بخشی از پیاده‌سازی PID Namespace در namespace.c
int setup_pid_namespace() {
    // namespace PID قبلاً با فراخوانی clone ایجاد شده است
    // نصب procfs برای نمایش صحیح PIDها
    if (mount("proc", "/proc", "proc", 0, NULL) != 0) {
        log_error("خطا در نصب /proc");
        return -1;
    }
    return 0;
}
```

۲. Mount-Namespace : امکان ایزوله‌سازی نقاط اتصال (mount-points) را فراهم می‌کند:

```
int setup_mount_namespace() {
    // ایزوله کردن mount namespace با تنظیم MS_PRIVATE
    if (mount(NULL, "/", NULL, MS_REC | MS_PRIVATE, NULL) != 0) {
        log_error("خطا در تنظیم MS_PRIVATE برای /");
        return -1;
    }
    return 0;
}
```

۳. UTS-Namespace : امکان داشتن hostname جداگانه برای هر کانتینر:

```
int setup_uts_namespace(const char *hostname) {
    // تنظیم hostname برای کانتینر
    if (sethostname(hostname, strlen(hostname)) != 0) {
        log_error("خطا در تنظیم hostname");
        return -1;
    }
    return 0;
}
```

۴. User-Namespace : امکان نگاشت شناسه‌های کاربر و گروه بین کانتینر و میزبان:

```
int setup_user_namespace() {
    // نگاشت UID به (0) داخل کانتینر UID خارج از کانتینر
    int uid_map_fd = open("/proc/self/uid_map", O_WRONLY);
    if (uid_map_fd == -1) {
        log_error("خطا در باز کردن /proc/self/uid_map");
        return -1;
    }

    char uid_map[100];
    snprintf(uid_map, sizeof(uid_map), "0 %d 1", getuid());
    if (write(uid_map_fd, uid_map, strlen(uid_map)) != strlen(uid_map)) {
        log_error("خطا در نوشتن به uid_map");
        close(uid_map_fd);
        return -1;
    }
    close(uid_map_fd);

    // مشابه همین کار برای GID
    // ...
    return 0;
}
```

۵. Network-Namespace : ایجاد استک شبکه مستقل برای هر کانتینر

۶. IPC-Namespace : ایزوله‌سازی ارتباطات بین‌پروسه‌ای

ایجاد namespace ها با استفاده از system-call-clone در زمان ایجاد فرآیند کانتینر انجام می‌شود:

```
pid_t pid = clone(container_process, stack + stack_size,
                  CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWUTS |
                  CLONE_NEWUSER | CLONE_NEWNET | CLONE_NEWIPC, config);
```

۲.۳ cgroup ها

Control-Groups (cgroups) امکان محدودسازی، حسابرسی و ایزوله‌سازی مصرف منابع فرآیندها را فراهم می‌کنند. SimpleContainer از cgroup ۲v استفاده می‌کند که نسل جدید این سیستم است.

۱. راه‌اندازی cgroup :

```
int cgroup_setup(container_config_t *config) {
    // اطمینان از وجود دایرکتوری پایه cgroup
    if (!directory_exists(CGROUP_BASE_PATH)) {
        if (create_directory(CGROUP_BASE_PATH, 0755) != 0) {
            log_error("خطا در ایجاد دایرکتوری پایه cgroup");
            return -1;
        }
    }

    // ساخت مسیر کامل cgroup برای این کانتینر
    char cgroup_full_path[512];
    snprintf(cgroup_full_path, sizeof(cgroup_full_path), "%s/%s", CGROUP_BASE_PATH, config->id);

    // ایجاد دایرکتوری cgroup برای کانتینر
    if (create_directory(cgroup_full_path, 0755) != 0) {
        log_error("خطا در ایجاد دایرکتوری cgroup برای کانتینر");
        return -1;
    }

    return 0;
}
```

۲. محدودیت حافظه:

```
int cgroup_set_memory_limit(container_config_t *config, uint64_t limit_bytes) {
    char cgroup_full_path[512];
    snprintf(cgroup_full_path, sizeof(cgroup_full_path), "%s/%s", CGROUP_BASE_PATH, config->id);

    // تنظیم محدودیت حافظه
    char limit_str[32];
    snprintf(limit_str, sizeof(limit_str), "%lu", limit_bytes);

    if (write_cgroup_file(cgroup_full_path, "memory.max", limit_str) != 0) {
        log_error("خطا در تنظیم محدودیت حافظه");
        return -1;
    }

    return 0;
}
```

۳. تخصیص CPU :

```
int cgroup_set_cpu_affinity(container_config_t *config, int cpu_id) {
    char cgroup_full_path[512];
    snprintf(cgroup_full_path, sizeof(cgroup_full_path), "%s/%s", CGROUP_BASE_PATH, config->id);

    // ساخت ماسک CPU
    char cpuset_str[256] = {0};
    snprintf(cpuset_str, sizeof(cpuset_str), "%d", cpu_id);

    if (write_cgroup_file(cgroup_full_path, "cpuset.cpus", cpuset_str) != 0) {
        log_error("خطا در تنظیم تخصیص CPU");
        return -1;
    }

    return 0;
}
```

۴. محدودیت I/O :

```
int cgroup_set_io_weight(container_config_t *config, uint64_t weight) {
    char cgroup_full_path[512];
    snprintf(cgroup_full_path, sizeof(cgroup_full_path), "%s/%s", CGROUP_BASE_PATH, config->id);

    // تنظیم وزن I/O
    char weight_str[32];
    snprintf(weight_str, sizeof(weight_str), "%lu", weight);

    if (write_cgroup_file(cgroup_full_path, "io.weight", weight_str) != 0) {
        log_error("خطا در تنظیم وزن I/O");
        return -1;
    }

    return 0;
}
```

۵. نظارت بر مصرف منابع:

```
int cgroup_get_memory_usage(container_config_t *config, uint64_t *usage) {
    char cgroup_full_path[512];
    snprintf(cgroup_full_path, sizeof(cgroup_full_path), "%s/%s", CGROUP_BASE_PATH, config->id);

    char buffer[128];
    if (read_cgroup_file(cgroup_full_path, "memory.current", buffer, sizeof(buffer)) != 0) {
        log_error("خطا در خواندن مصرف حافظه");
        return -1;
    }

    *usage = strtoull(buffer, NULL, 10);
    return 0;
}
```

۴ محدودیت‌ها و چالش‌ها

در طول توسعه SimpleContainer با چالش‌ها و محدودیت‌های متعددی مواجه شدیم:

محدودیت‌های User-Namespace : پیاده‌سازی نگاشت UID/GID بین کانتینر و میزبان پیچیدگی‌های خاصی داشت، به خصوص در سیستم‌هایی که Namespace User به صورت پیش‌فرض غیرفعال است.

مدیریت شبکه: پیاده‌سازی کامل مدیریت شبکه (port mapping، NAT، DNS) بسیار پیچیده بود و در نسخه فعلی محدود شده است.

سازگاری cgroup-v۲ : برخی توزیع‌های لینوکس هنوز به صورت کامل از cgroup-v۲ پشتیبانی نمی‌کنند، که منجر به مشکلات سازگاری می‌شود.

پایداری overlayfs : در برخی موارد، به خصوص با فایل‌های بزرگ، عملکرد overlayfs دچار مشکل می‌شد.

پیچیدگی‌های eBPF : پیاده‌سازی مانیتورینگ کامل با eBPF نیازمند دانش عمیق از کرنل لینوکس است و در نسخه فعلی به صورت محدود پیاده‌سازی شده است.

۵ تست و بررسی کارایی

به طور مفصل کارایی برنامه داخل فایل README توضیح داده شده است.