دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر
پروژه سیستم‌های عامل

عنوان:

پیاده‌سازی سیستم build در docker

نگارش

مهبد خالتی ۴۰۲۱۰۵۹۴۳
پوریا رحمانی ۴۰۲۱۱۱۴۱۸
نیما قدیرنیا ۴۰۲۱۱۱۳۲۳
متین غیاثی ۴۰۲۱۰۶۲۲۹

بهمن ۱۴۰۴

فهرست مطالب

۱ مقدمه ۵

۲ پیاده‌سازی ۶

۲-۱ مروری بر ساختار کلی پروژه ۶

۲-۲ تحلیل و توضیح فایل utils.c ۸

۲-۲-۱ توابع مربوط به الگوریتم هش FNV-1a ۸

۲-۲-۲ هش فایل و مسیرها ۹

۲-۲-۳ مدیریت UUID ۱۰

۲-۲-۴ بررسی و ایجاد مسیرها ۱۰

۲-۲-۵ کپی بازگشتی فایل‌ها ۱۱

۲-۲-۶ محاسبه اندازه دایرکتوری ۱۱

۲-۲-۷ حذف بازگشتی ۱۱

۲-۲-۸ مدیریت و نرمال‌سازی مسیرها ۱۲

۲-۲-۹ توابع کمکی رشته‌ای ۱۲

۲-۳ تحلیل و توضیح فایل image_store.c ۱۳

۲-۳-۱ sanitize_component ۱۳

۲-۳-۲ parse_image_ref ۱۳

۲-۳-۳ image_meta_path_from_ref ۱۴

۲-۳-۴ save_image_meta ۱۵

۲-۳-۵ load_image_meta ۱۶

۲-۳-۶ image_exists ۱۶

۲-۳-۷ layer_exists ۱۶

۲-۳-۸ read_layer_link ۱۷

۲-۳-۹ layer_mount_entry_from_id ۱۷
۲-۳-۱۰ extract_layer_id_from_diff_entry ۱۸
۲-۳-۱۱ normalize_chain_entry ۱۹
۲-۳-۱۲ append_chain_entry ۱۹
۲-۳-۱۳ normalize_chain ۲۰
۲-۳-۱۴ layer_chain_from_top ۲۰
۲-۳-۱۵ resolve_zocker_image_chain ۲۱
۲-۳-۱۶ register_layer_cache ۲۲
۲-۳-۱۷ lookup_layer_cache ۲۲
۲-۳-۱۸ مدیریت کش لایه‌ها ۲۳
۲-۳-۱۹ توابع نمایش و لیست‌کردن ایمیج‌ها ۲۳
۲-۳-۲۰ توابع مجموعه رشته (String Set) ۲۳
۲-۳-۲۱ توابع جستجو و نشانه‌گذاری ۲۴
۲-۳-۲۲ توابع پاکسازی (Pruning) ۲۴
۲-۴ تحلیل و توضیح فایل config.c ۲۵
۲-۴-۱ گسترش لیست Subcommandها ۲۵
۲-۴-۲ تفکیک اعتبارسنجی بر اساس نوع دستور ۲۵
۲-۴-۳ سازگاری با دستورات جدید پروژه ۲۶
۲-۵ تحلیل و توضیح فایل setup.c ۲۶
۲-۵-۱ اضافه شدن تابع ensure_dir ۲۶
۲-۵-۲ گسترش تابع setup_zocker_dir ۲۶
۲-۵-۳ اضافه شدن مکانیزم Resolve کردن Base Image ۲۷
۲-۵-۴ بازسازی زنجیره لایه‌های Docker ۲۷
۲-۵-۵ اضافه شدن اعتبارسنجی مسیرهای Overlay ۲۷

۲-۵-۶ تغییر در امضای setup_container_dir ۲۸
۲-۶ تحلیل و توضیح فایل build.c ۲۸
۲-۶-۱ ثابت‌ها و هدرها ۲۸
۲-۶-۲ ساختار struct arg_map ۲۸
۲-۶-۳ ساختار struct stage_ctx ۲۹
۲-۶-۴ تابع arg_map_set ۲۹
۲-۶-۵ تابع arg_map_get ۲۹
۲-۶-۶ تابع arg_map_copy ۳۰
۲-۶-۷ تابع init_cli_args_map ۳۰
۲-۶-۸ تابع substitute_args ۳۰
۲-۶-۹ تابع make_temp_dir ۳۰
۲-۶-۱۰ تابع compute_state_hash ۳۱
۲-۶-۱۱ تابع resolve_stage_chain ۳۱
۲-۶-۱۲ تابع run_in_chroot ۳۱
۲-۶-۱۳ تابع basename_of ۳۲
۲-۶-۱۴ تابع copy_into_rootfs ۳۲
۲-۶-۱۵ تابع mount_overlay ۳۲
۲-۶-۱۶ تابع with_stage_snapshot ۳۲
۲-۶-۱۷ تابع create_layer_dirs ۳۳
۲-۶-۱۸ ساختار struct run_apply_ctx و تابع apply_run_layer ۳۳
۲-۶-۱۹ ساختار struct workdir_apply_ctx و تابع apply_workdir_layer ۳۳
۲-۶-۲۰ ساختار struct copy_apply_ctx و تابع apply_copy_layer ۳۳
۲-۶-۲۱ ساختار struct add_apply_ctx و توابع مرتبط ۳۴
۲-۶-۲۲ تابع apply_noop_layer ۳۴

۲-۶-۲۳ تابع create_layer ۳۴
۲-۶-۲۴ توابع مربوط به پارس کردن Zockerfile ۳۶
۲-۶-۲۵ تابع build_image_from_config ۳۷
۲-۷ توضیح فایل main.c ۴۰
۲-۷-۱ تابع append_run_command ۴۰
۲-۷-۲ تابع parse_build_arg_value ۴۰
۲-۷-۳ حالت‌بندی روی token های دستور کاربر ۴۰
۲-۷-۴ حالت بندی روی نوع دستور ۴۱

۳ تست ۴۱
۳-۱ مراحل پیش‌نیاز و آماده‌سازی محیط ۴۱
۳-۲ آماده‌سازی RootFS مینیمال ۴۲
۳-۳ سناریوهای آزمون ۴۲
۳-۳-۱ آزمون اول: بررسی مکانیزم Cache و لایه‌بندی ۴۲
۳-۳-۲ آزمون دوم: ساخت چندمرحله‌ای (Multi-stage Build) ۴۳
۳-۴ تست‌های دستی ۴۴

۱ مقدمه

در این پروژه، هدف طراحی و پیاده‌سازی یک ابزار مدیریت ایمیج و بیلد کانتینر با الهام از مکانیزم داخلی Docker است. تمرکز اصلی پروژه بر درک عمیق مفاهیم لایه‌بندی (Layered Architecture)، سیستم‌فایل‌های ترکیبی (OverlayFS)، مکانیزم‌های caching در فرآیند build و مدیریت هش (Hash-based Identification) قرار دارد.

در معماری Docker، هر ایمیج از مجموعه‌ای از لایه‌های فقط‌ـخواندنی (Read-only Layers) تشکیل شده است که به صورت سلسله‌مراتبی روی یکدیگر قرار می‌گیرند. هر دستور در Dockerfile منجر به ایجاد یک لایه جدید می‌شود. این طراحی باعث می‌شود:

- استفاده مجدد از لایه‌های قبلی امکان‌پذیر باشد
- مکانیزم build cache به صورت کارآمد پیاده‌سازی شود
- مصرف فضای ذخیره‌سازی بهینه گردد

در این پروژه، ابزاری با نام `zocker` توسعه داده شده است که قابلیت‌های زیر را فراهم می‌کند:

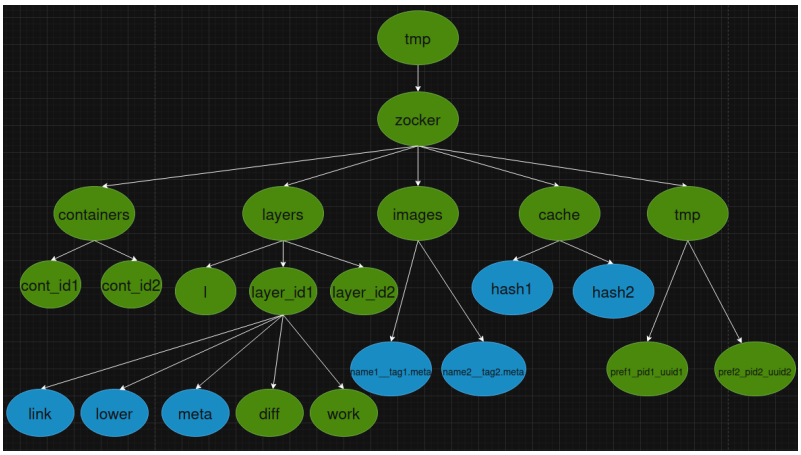
- ساخت ایمیج بر اساس فایل `Zockerfile`
- پشتیبانی از دستورات FROM, RUN, COPY, ADD, WORKDIR, ARG, CMD
- پیاده‌سازی معماری لایه‌ای با استفاده از OverlayFS
- تولید UUID یکتا برای هر لایه
- محاسبه hash هر مرحله بیلد بر اساس ترکیب hash لایه زیرین و دستور جاری
- پیاده‌سازی مکانیزم build caching
- نمایش تاریخچه لایه‌ها (history)
- حذف ایمیج‌ها و هرس لایه‌های بلااستفاده (prune)

۲ پیاده‌سازی

۲-۱ مروری بر ساختار کلی پروژه

در این پروژه که بر پایه تمرین هشتم نوشته شده، سه فایل utils.c ،image_store.c و build.c نیز به همراه header هایشان به پروژه اضافه شده‌اند که در بخش‌های بعدی به توضیح آن‌ها خواهیم پرداخت.

در این پروژه نیز مثل تمرین‌ها، پوشه اصلی zocker پوشه tmp/zocker خواهد بود. ساختار درختی کلی ذخیره فایل‌ها به صورت زیر خواهد بود:



- پوشه containers مانند قبل است و برای نگهداری کانتینرها به‌کار می‌رود.
- پوشه layers شامل پوشه l است که حاوی symlink هاست و همینطور به‌ازای هر لایه‌ی ساخته شده، یک uuid مطابق آنچه در توضیح توابع فایل utils.c می‌آید، تولید می شود و به‌عنوان شناسه لایه در نظر گرفته می‌شود و یک پوشه برای آن‌لایه با آن اسم ساخته می‌شود.
- در پوشه مربوط به هر لایه، یک پوشه diff داریم که تغییرات نسبت به لایه parent را نشان می‌دهند و symlink ها هم به همین پوشه‌ها اشاره می‌کنند.
  یک پوشه work هم برای محاسبات درونی overlay ساخته می‌شود.
  یک فایل lower هم داریم که آدرس لایه‌های پایین‌تر را (در واقع آدرس ابسترکت symlink مربوط به آن‌ها را) به صورت یک لیست که با : از هم جدا شده‌است، به‌ترتیب به‌طوریکه چپ‌ترین آدرس مربوط به بالاترین لایه زیر لایه کنونی است، نگه می‌دارد. و یک فایل link که شناسه کوتاهتری را که از روی id می‌سازیم و قرار است به عنوان symlink استفاده شود را در آن ذخیره می‌کنیم.

یک فایل meta هم داریم که متادیتای مربوط به لایه (مثل سایز و پوشه کاری و ...) را در خود به صورت key-value ذخیره می‌کند.

- در پوشه images فایل‌های metadata مربوط به image ها را به صورت key-value نگه می‌داریم. این اطلاعات شامل نام و تگ ایمیج، لایه بالایی ایمیج و ... می‌شود. بنابراین با نگه داری این فایل، می‌توانیم از روی فایل lower لایه بالایی، می‌توانیم به سایر لایه‌ها نیز دسترسی داشته‌باشیم.
  فرمت نام این فایل‌ها به صورت name___tag.meta است.

در توضیح فایل build.c توضیح خواهیم داد که هر دستور یک descriptor دارد که در تولید hash یک لایه، همه آن descriptor ها را در کنار هم می‌آوریم (از کاراکتر | به‌عنوان کاراکتر جداکننده استفاده می‌کنیم).

- در پوشه cache، به ازای هر لایه موجود، یک پوشه با نام hash آن لایه وجود دارد که محتوای آن id آن لایه است. برای بررسی این‌که یک لایه وجود دارد یا خیر، از این پوشه استفاده می‌کنیم و hash لایه‌ای که قرار است ساخته شود را از پیش محاسبه می‌کنیم (چون hash تنها بر اساس descriptor دستورهاست، محاسبه hash به سرعت قابل انجام است.)
  سپس این hash را با نام تمامی پوشه‌های درون پوشه cache مقایسه می‌کنیم. وجود پوشه‌ای با نام آن hash به منزله cache hit تلقی می شود.

از آنجا که ایمیج‌ها تنها انباشته‌ای از اطلاعات هستند و توتن پردازشی ندارند، برای اینکه بتوانیم با استفاده از دستوراتی مثل COPY, ADD یا RUN لایه‌های جدیدی برای ایمیج درحال ساخت خود ایجاد کنیم، باید یک کانتینر موقت بسازیم و بعد از اتمام کارمان نابودش کنیم!برای همین، په پوشه tmp نیاز داریم.

- داخل پوشه tmp پوشه‌هایی با فرمت اسم prefix_pid_uuid وجود دارند.
  prefix های ممکن، build, add, snapshot هستند. pid همان شناسه پردازه کنونی است و uuid هم با تابعی که در بخش utils.c توضیح می‌دهیم ساخته می‌شود.
  برای کلمه ADD صرفا فایل مدنظر با اسم download.bin در پوشه ساخته شده دانلود می شود و سپس در مقصد مدنظر کپی می‌شود.
  برای پیشوند snapshot که در multi-stage-building برای کارهایی مثل copy کردن استفاده می‌شود ، درون پوشه موقتی که می‌سازیم، همه پوشه‌های merged و work و upper را ایجاد

می‌کنیم تا مثل یک کانتینر موقت عمل کند تا بتوانیم عملیات موردنظر را بدرستی انجام دهیم. برای پیشوند build که در زمان ساختن لایه جدید برای ایمیجمان استفاده می‌شود، فقط پوشه merged را در پوشه موقت می‌سازیم و پوشه‌های work و upper را که باید در لایه به‌طور دائمی ذخیره شوند، در خود پوشه لایه می‌سازیم. (توجه کنید پوشه diff در پوشه مربوط به لایه، همان کار upper را خواهد کرد) و با استفاده از container موقت ایجاد شده، کار خود را می‌کنیم.

شیوه پیاده‌سازی دستورات rmi ، prune ، history ، و images در توضیحات مربوط به فایل image_store.c آمده‌است. (می‌توانید ابتدا در توضیحات فایل main.c ببینید به ازای هر دستور، چه تابعی صدا زده شده است.)
برای دستور build هم تابع build_image_from_config صدا زده‌شده است که آخرین و مفصل‌ترین تابع build.c است.
اینکه به ازای هر کلمه خاص در Zockerfile چه می‌کنیم هم در تابع build_image_from_config مشخص می‌شود. به طور کلی، برای دستوراتی که اجرایشان منجر به ایجاد یک لایه جدید برای image می‌شود، یک struct و یک تابع با امضای مشابه تعریف کردیم تا با پاس دادن یک func-tion pointer به همراه struct مد نظر به تابع create_layer مجبور نشویم برای هر کلمه یک تابع create_layer جدا بزنیم و با یک تابع جامع و تعدادی تابع جزئی برای هر دستور پیاده‌سازی ساختن لایه جدید را انجام دهیم.

۲-۲ تحلیل و توضیح فایل `utils.c`

فایل `utils.c` شامل مجموعه‌ای از توابع زیرساختی است که عملیات حیاتی مرتبط با مدیریت لایه‌ها، سیستم فایل، محاسبه هش و مکانیزم کش را پیاده‌سازی می‌کند. این فایل ستون فقرات منطقی پروژه محسوب می‌شود. در ادامه، تمامی توابع این فایل به تفکیک معرفی و تحلیل می‌شوند.

۲-۲-۱ توابع مربوط به الگوریتم هش FNV-1a

تابع `fnv1a_init()`:
این تابع مقدار اولیه (Offset Basis) الگوریتم FNV-1a را بازمی‌گرداند. این مقدار نقطه شروع محاسبه هش بوده و ثابت استاندارد الگوریتم است.

تابع `fnv1a_update(uint64_t hash, const void *data, size_t len)`:
تابع اصلی پردازش داده در الگوریتم FNV-1a است. این تابع:

- داده ورودی را به صورت بایتی پیمایش می‌کند.
- هر بایت را با مقدار هش XOR می‌کند.
- حاصل را در عدد اول FNV ضرب می‌کند.

این عملیات باعث تولید هش یکنواخت و مقاوم در برابر برخورد (collision) نسبی می‌شود.

تابع `fnv1a_hex(uint64_t hash, char out[17])`:
مقدار هش ۶۴ بیتی را به رشته هگزادسیمال ۱۶ کاراکتری تبدیل می‌کند تا برای ذخیره در متادیتا و مقایسه در مکانیزم کش استفاده شود.

تابع `hash_string(const char *s, char out_hex[17])`
یک رشته (مانند دستور RUN یا COPY) را هش کرده و خروجی هگزادسیمال تولید می‌کند. این تابع در محاسبه hash مرحله build استفاده می‌شود.

۲-۲-۲ هش فایل و مسیرها

تابع `hash_file_content(const char *path, char out_hex[17])`:
محتوای فایل را به صورت بلوک‌های ثابت (معمولاً 8KB) خوانده و هش کامل فایل را محاسبه می‌کند. این تابع برای تشخیص تغییر محتوای فایل‌ها در COPY/ADD کاربرد دارد.

تابع `hash_path_internal(const char *path, uint64_t *hash)`:
تابع بازگشتی برای محاسبه هش یک مسیر است. رفتار آن بسته به نوع فایل متفاوت است:

- اگر فایل معمولی باشد: اندازه و محتوای آن در هش لحاظ می‌شود.
- اگر دایرکتوری باشد: نام فرزندان مرتب شده و بازگشتی هش می‌شوند.
- اگر symbolic link باشد: مقصد لینک هش می‌شود.

این طراحی تضمین می‌کند که هر تغییری در ساختار یا محتوای مسیر باعث تغییر hash شود.

تابع hash_path_recursive(const char *path, char out_hex[17]):
رابط عمومی برای محاسبه هش کامل یک مسیر. از تابع داخلی استفاده کرده و نتیجه را به صورت رشته هگزادسیمال بازمی‌گرداند.

این توابع مستقیماً در پیاده‌سازی build cache نقش کلیدی دارند.

۲-۲-۳ مدیریت UUID

تابع generate_uuid(char *uuid_str):
این تابع یک شناسه یکتا برای هر لایه تولید می‌کند.برای این کار ابتدا تلاش می‌شود آیدی مدنظر از مسیر /proc/sys/kernel/random/uuid خوانده شود. در صورت عدم موفقیت، با استفاده از زمان سیستم و داده تصادفی UUID تولید می‌شود.
هر لایه OverlayFS دارای یک UUID مستقل است که در ساختار پروژه ذخیره می‌شود.

۲-۲-۴ بررسی و ایجاد مسیرها

تابع path_exists(const char *path):
بررسی می‌کند که آیا مسیر مشخص شده وجود دارد یا خیر.

تابع is_directory(const char *path):
با استفاده از stat نوع فایل را بررسی کرده و مشخص می‌کند که آیا دایرکتوری است یا خیر.

تابع ensure_dir_exists(const char *path):
اگر دایرکتوری وجود نداشته باشد، آن را ایجاد می‌کند.

تابع ensure_parent_dirs(const char *path):
برای مسیرهای تو در تو (nested paths) تمامی دایرکتوری‌های والد را ایجاد می‌کند. این تابع برای اجرای صحیح COPY و ADD ضروری است.

۲-۲-۵ کپی بازگشتی فایل‌ها

تابع `copy_file_data(const char *src, const char *dst)`:
محتوای فایل مبدا را با استفاده از system call های `open`، `read`، `write` به مقصد منتقل می‌کند و مجوزهای مناسب را تنظیم می‌کند.

تابع `copy_path_recursive(const char *src, const char *dst)`:
یک مسیر کامل را بازگشتی کپی می‌کند:

- برای دایرکتوری‌ها: ابتدا مقصد ساخته شده و سپس محتویات کپی می‌شوند.
- برای فایل‌ها: از copy_file_data استفاده می‌شود.
- برای symbolic link ها: لینک مجدداً ایجاد می‌شود.

این تابع در اجرای دستورات COPY و ADD استفاده می‌شود.

۲-۲-۶ محاسبه اندازه دایرکتوری

تابع `dir_size_internal(const char *path, size_t *total)`:
به صورت بازگشتی اندازه تمام فایل‌های داخل مسیر را جمع‌آوری می‌کند.

تابع `dir_size_bytes(const char *path)`:
رابط عمومی برای محاسبه اندازه کل یک لایه. در دستور history برای نمایش حجم هر لایه استفاده می‌شود.

۲-۲-۷ حذف بازگشتی

تابع `remove_recursive(const char *path)`:
یک مسیر را به صورت کامل حذف می‌کند:

- ابتدا محتویات دایرکتوری حذف می‌شود
- سپس خود دایرکتوری با `rmdir` حذف می‌شود

- برای فایل‌ها از `unlink` استفاده می‌شود

این تابع در پیاده‌سازی دستور `rmi zocker` و `prune zocker` استفاده می‌شود.

۲-۲-۸ مدیریت و نرمال‌سازی مسیرها

تابع `join_paths(const char *base, const char *child)`:
دو مسیر را به صورت ایمن ترکیب می‌کند و از ایجاد اسلش اضافی جلوگیری می‌کند.

تابع `normalize_abs_path(const char *path, char *out)`:
مسیر مطلق را نرمال‌سازی می‌کند:

- حذف "."
- پردازش ".."
- جلوگیری از خروج از ریشه (path traversal)

تابع `normalize_container_path(...)`:
مسیرهای داخل کانتینر را بر اساس working directory تنظیم و نرمال می‌کند.

۲-۲-۹ توابع کمکی رشته‌ای

تابع `trim_whitespace(char *str)`:
فاصله‌های ابتدا و انتهای رشته را حذف می‌کند. در parsing دستورات Zockerfile کاربرد دارد.

تابع `starts_with(const char *str, const char *prefix)`:
بررسی می‌کند که آیا رشته با پیشوند مشخصی شروع می‌شود یا خیر.

تابع `ends_with(const char *str, const char *suffix)`:
بررسی می‌کند که آیا رشته با پسوند مشخصی پایان می‌یابد یا خیر.

۲-۳ تحلیل و توضیع فایل image_store.c

این فایل مسئول مدیریت تصاویر کانتینر، لایه‌ها و metadata مربوط به آنها می‌باشد. در این فایل، از OverlayFS برای پیاده‌سازی سیستم لایه‌ای تصاویر استفاده شده است.

۲-۳-۱ sanitize_component

```
static int sanitize_component(const char *src,
                              char *dst,
                              size_t dst_size)
```

هدف: این تابع یک رشته ورودی را پاکسازی و ایمن‌سازی می‌کند تا فقط شامل کاراکترهای مجاز باشد.

عملکرد:

- تمام کاراکترهای ورودی را بررسی می‌کند
- فقط حروف الفبا، اعداد، خط تیره (-)، زیرخط (_) و نقطه (.) را مجاز می‌داند
- سایر کاراکترها را با زیرخط جایگزین می‌کند
- در صورت موفقیت 0 و در صورت خطا 1 برمی‌گرداند

کاربرد: مطمئن شدن از اینکه نام فایل‌ها و مسیرهایی که از نام و تگ تصویر ساخته می‌شوند، ایمن و بدون کاراکترهای خطرناک هستند.

۲-۳-۲ parse_image_ref

```
int parse_image_ref(const char *ref,
                    char *name, size_t name_size,
                    char *tag, size_t tag_size)
```

هدف: تجزیه یک reference تصویر به دو بخش نام و تگ.

عملکرد:

- reference تصویر را به فرمت name:tag تجزیه می‌کند
- آخرین دونقطه (:) را پیدا می‌کند و تصویر را بر اساس آن تقسیم می‌نماید
- اگر slash (/) بعد از دونقطه باشد، دونقطه را نادیده می‌گیرد
- اگر تگی مشخص نشده باشد، به طور پیش‌فرض تگ را latest قرار می‌دهد
- نتایج را در متغیرهای name و tag ذخیره می‌کند

مثال: برای alpine:3.14 نتیجه name=alpine و tag=3.14 خواهد بود.

۲-۳-۳ image_meta_path_from_ref

```
static int image_meta_path_from_ref(const char *ref,
                                    char *path,
                                    size_t path_size)
```

هدف: ساخت مسیر فایل metadata یک تصویر بر اساس reference آن.

عملکرد:

- ابتدا reference را با استفاده از parse_image_ref تجزیه می‌کند
- نام و تگ را با استفاده از sanitize_component ایمن‌سازی می‌کند
- مسیر فایل را به صورت <ZOCKER_IMAGES_DIR>/name___tag.meta می‌سازد
- از ___ برای جدا کردن نام از تگ استفاده می‌کند

مثال: برای alpine:3.14 مسیر خروجی به صورت /path/to/images/alpine___3.14.meta خواهد بود.

load_image_meta_from_path

```
static int load_image_meta_from_path(const char *path,
                                     struct image_meta *meta)
```

هدف: بارگذاری اطلاعات metadata یک تصویر از فایل مشخص شده.

عملکرد:

- فایل metadata را باز کرده و خط به خط می‌خواند
- هر خط به صورت key=value تجزیه می‌شود
- کلیدهایی که پشتیبانی می‌شوند: name، tag، ref، top_layer، created_at، cmd
- مقادیر را در ساختار image_meta ذخیره می‌کند
- فضاهای خالی اضافی را با استفاده از trim_whitespace حذف می‌کند

نکته: این تابع داخلی است و مستقیماً از بیرون فراخوانی نمی‌شود.

۲-۳-۴ save_image_meta

```
int save_image_meta(const struct image_meta *meta)
```

هدف: ذخیره اطلاعات metadata یک تصویر در فایل مربوطه.

عملکرد:

- اگر فیلد ref خالی باشد، آن را از name و tag می‌سازد
- اگر تگ خالی باشد، از latest استفاده می‌کند
- مسیر فایل metadata را با استفاده از image_meta_path_from_ref می‌سازد
- اگر created_at خالی باشد، زمان فعلی را به عنوان timestamp قرار می‌دهد
- تمام فیلدها را به صورت key=value در فایل می‌نویسد

فیلدهای ذخیره شده: name، tag، ref، top_layer، created_at، cmd

load_image_meta ۵-۳-۲

```
int load_image_meta(const char *ref,
                    struct image_meta *meta)
```

هدف: بارگذاری metadata یک تصویر بر اساس reference آن.

عملکرد:

- مسیر فایل metadata را با استفاده از image_meta_path_from_ref می‌سازد
- سپس load_image_meta_from_path را برای خواندن فایل فراخوانی می‌کند
- اگر فایل وجود نداشته باشد یا قابل خواندن نباشد، خطا برمی‌گرداند

کاربرد: این تابع عمومی است و از سایر بخش‌های برنامه برای دریافت اطلاعات تصویر استفاده می‌شود.

image_exists ۶-۳-۲

```
int image_exists(const char *ref)
```

هدف: بررسی اینکه آیا یک تصویر با reference مشخص شده وجود دارد یا خیر.

عملکرد:

- مسیر فایل metadata تصویر را محاسبه می‌کند
- با استفاده از path_exists وجود فایل را بررسی می‌کند
- اگر فایل وجود داشته باشد 1 و در غیر این صورت 0 برمی‌گرداند

کاربرد: قبل از عملیات‌هایی مانند pull یا run برای بررسی وجود تصویر استفاده می‌شود.

layer_exists ۷-۳-۲

```
int layer_exists(const char *layer_id)
```

هدف: بررسی اینکه آیا یک لایه با شناسه مشخص شده وجود دارد یا خیر.

عملکرد:

- مسیر پوشه لایه را به صورت <ZOCKER_LAYERS_DIR>/layer_id می‌سازد
- با استفاده از is_directory بررسی می‌کند که آیا این مسیر یک دایرکتری معتبر است
- اگر دایرکتری وجود داشته باشد 1 و در غیر این صورت 0 برمی‌گرداند

کاربرد: برای تأیید وجود لایه‌ها قبل از استفاده از آنها در mount یا سایر عملیات.

۸-۳-۲ read_layer_link

```
static int read_layer_link(const char *layer_id,
                           char *out_link,
                           size_t out_link_size)
```

هدف: خواندن محتوای فایل link یک لایه.

عملکرد:

- مسیر فایل link را به صورت <ZOCKER_LAYERS_DIR>/layer_id/link می‌سازد
- محتوای فایل را می‌خواند (که شامل شناسه کوتاه یا short ID لایه است)
- کاراکترهای پایان خط را حذف می‌کند
- در صورت موفقیت 0 و در صورت خطا 1 برمی‌گرداند

نکته: فایل link حاوی یک شناسه کوتاه است که برای ساخت symlink در دایرکتری l استفاده می‌شود.

۹-۳-۲ layer_mount_entry_from_id

```
static int layer_mount_entry_from_id(const char *layer_id,
                                     char *out,
                                     size_t out_size)
```

هدف: ساخت مسیر mount entry برای یک لایه.

عملکرد:

- ابتدا سعی می‌کند link لایه را بخواند
- اگر link وجود داشته باشد، مسیر به صورت <ZOCKER_LAYER_LINKS_DIR>/link_id ساخته می‌شود
- اگر link وجود نداشته باشد، مسیر مستقیم diff به صورت <ZOCKER_LAYERS_DIR>/layer_id/diff استفاده می‌شود
- این مسیر برای استفاده در lowerdir یا upperdir در OverlayFS مورد نیاز است

کاربرد: تعیین مسیر صحیح برای mount کردن لایه‌ها در OverlayFS.

۲-۳-۱۰ extract_layer_id_from_diff_entry

```
static int extract_layer_id_from_diff_entry(
    const char *entry,
    char *out_layer_id,
    size_t out_layer_id_size)
```

هدف: استخراج شناسه لایه از یک entry که به صورت مسیر diff است.

عملکرد:

- بررسی می‌کند که entry با پیشوند <ZOCKER_LAYERS_DIR>/ شروع شود
- قسمت بین دایرکتری layers و diff/ را به عنوان layer_id استخراج می‌کند
- اعتبارسنجی می‌کند که entry با diff/ پایان یابد
- شناسه استخراج شده را در out_layer_id ذخیره می‌کند

مثال: از /path/layers/abc123/diff شناسه abc123 استخراج می‌شود.

normalize_chain_entry ۱۱-۳-۲

```
static int normalize_chain_entry(const char *entry,
                                 char *out,
                                 size_t out_size)
```

هدف: نرمال‌سازی یک entry در زنجیره لایه‌ها.

عملکرد:

- اگر entry از پوشه ZOCKER_LAYER_LINKS_DIR باشد، آن را بدون تغییر برمی‌گرداند
- اگر entry یک مسیر diff باشد، شناسه لایه را استخراج می‌کند
- سپس مسیر mount entry صحیح را با استفاده از layer_mount_entry_from_id می‌سازد
- اگر هیچ تبدیلی نیاز نباشد، entry را بدون تغییر کپی می‌کند

کاربرد: تضمین اینکه تمام entries در زنجیره لایه‌ها به فرمت استانداردی هستند که برای OverlayFS قابل استفاده است.

append_chain_entry ۱۲-۳-۲

```
static int append_chain_entry(char *chain,
                              size_t chain_size,
                              const char *entry,
                              int with_separator)
```

هدف: افزودن یک entry به انتهای رشته زنجیره لایه‌ها.

عملکرد:

- طول فعلی رشته chain را محاسبه می‌کند
- فضای باقی‌مانده را بررسی می‌کند
- اگر with_separator برابر 1 باشد، دونقطه (:) را قبل از entry اضافه می‌کند

- در غیر این صورت entry را مستقیماً اضافه می‌کند
- از buffer overflow جلوگیری می‌کند

کاربرد: ساخت رشته lowerdir برای OverlayFS که شامل مسیرهای چندین لایه جدا شده با دونقطه است.

۱۳-۳-۲ normalize_chain

```
static int normalize_chain(const char *chain,
                           char *out,
                           size_t out_size)
```

هدف: نرمال‌سازی کل زنجیره لایه‌ها.

عملکرد:

- رشته chain ورودی را کپی می‌کند (چون strtok_r رشته را تغییر می‌دهد)
- زنجیره را با استفاده از دونقطه (:) به قسمت‌های مختلف تقسیم می‌کند
- هر entry را با normalize_chain_entry نرمال می‌کند
- تمام entries نرمال شده را با استفاده از append_chain_entry به هم متصل می‌کند
- رشته خروجی نرمال شده را برمی‌گرداند

کاربرد: آماده‌سازی زنجیره کامل لایه‌ها برای استفاده در OverlayFS mount.

۱۴-۳-۲ layer_chain_from_top

```
int layer_chain_from_top(const char *layer_id,
                         char *out_chain,
                         size_t out_size)
```

هدف: ساخت زنجیره کامل لایه‌ها از یک لایه بالایی (top layer).

عملکرد:

- بررسی می‌کند که لایه وجود دارد
- مسیر mount entry لایه بالایی را می‌سازد
- فایل lower لایه را می‌خواند که حاوی زنجیره لایه‌های زیرین است
- اگر فایل lower وجود داشته باشد، آن را نرمال‌سازی می‌کند
- لایه بالایی را با لایه‌های زیرین ترکیب می‌کند (به صورت top:lower1:lower2:...)
- اگر فایل lower وجود نداشته باشد، فقط لایه بالایی را برمی‌گرداند

کاربرد: ساخت رشته lowerdir کامل برای mount کردن تصویر با OverlayFS.

۱۵-۳-۲ resolve_zocker_image_chain

```
int resolve_zocker_image_chain(const char *ref,
                               char *out_chain,
                               size_t out_size)
```

هدف: حل زنجیره لایه‌های یک تصویر بر اساس reference آن.

عملکرد:

- metadata تصویر را با load_image_meta بارگذاری می‌کند
- شناسه لایه بالایی را از فیلد top_layer استخراج می‌کند
- layer_chain_from_top را فراخوانی می‌کند تا زنجیره کامل لایه‌ها را بسازد
- زنجیره نهایی را در out_chain برمی‌گرداند

کاربرد: این تابع برای دریافت زنجیره کامل لایه‌های یک تصویر استفاده می‌شود که برای mount کردن کانتینر ضروری است.

۲-۳-۱۶ register_layer_cache

```
int register_layer_cache(const char *hash,
                         const char *layer_id)
```

هدف: ثبت یک لایه در cache بر اساس hash آن.

عملکرد:

- مسیر فایل cache را به صورت <ZOCKER_CACHE_DIR>/hash می‌سازد
- فایل را باز کرده و شناسه لایه را در آن می‌نویسد
- این اطلاعات برای layer caching در build استفاده می‌شود
- در صورت موفقیت 0 و در صورت خطا 1 برمی‌گرداند

کاربرد: زمانی که یک لایه جدید ساخته می‌شود، hash آن (بر اساس دستور ساخت) ذخیره می‌شود تا در buildهای بعدی از آن استفاده شود و از ساخت مجدد جلوگیری شود.

۲-۳-۱۷ lookup_layer_cache

```
int lookup_layer_cache(const char *hash,
                       char *layer_id,
                       size_t layer_id_size)
```

هدف: جستجوی یک لایه در cache بر اساس hash.

عملکرد:

- مسیر فایل cache را به صورت <ZOCKER_CACHE_DIR>/hash می‌سازد
- اگر فایل وجود داشته باشد، شناسه لایه را از آن می‌خواند
- بررسی می‌کند که لایه با این شناسه واقعاً وجود دارد (با layer_exists)
- اگر لایه وجود داشته باشد، شناسه را برمی‌گرداند

- در صورت موفقیت 0 و در صورت عدم یافتن 1 برمی‌گرداند

کاربرد: در فرآیند build، قبل از اجرای هر دستور، بررسی می‌شود که آیا نتیجه آن دستور قبلاً cache شده است یا خیر. اگر cache موجود باشد، از اجرای مجدد دستور جلوگیری می‌شود.

۲-۳-۱۸ مدیریت کش لایه‌ها

- **register_layer_cache:** پس از ساخت یک لایه با موفقیت، شناسه (ID) آن را در فایلی که نام آن همان هش دستور (hash) است، ذخیره می‌کند.
- **lookup_layer_cache:** پیش از اجرای یک دستور ساخت، بررسی می‌کند آیا فایلی با نام هش مربوطه وجود دارد و لایه متناظر با آن هنوز در فایل‌سیستم موجود است یا خیر. در صورت وجود، لایه از کش خوانده می‌شود.

۲-۳-۱۹ توابع نمایش و لیست‌کردن ایمیج‌ها

- **format_age:** اختلاف زمان تولید یک ایمیج یا لایه را با زمان فعلی محاسبه کرده و به فرمت خوانا (ثانیه s، دقیقه m، ساعت h، و روز d) تبدیل می‌کند.
- **list_images:** دایرکتوری تصاویر (ZOCKER_IMAGES_DIR) را پیمایش کرده، تمامی فایل‌های meta. را خوانده و جدولی شامل نام ایمیج، بالاترین لایه و زمان ایجاد آن‌ها چاپ می‌کند.
- **print_image_history:** ساختار درختی یک ایمیج را از بالاترین لایه به پایین (سمت لایه پایه) با استفاده از ویژگی parent می‌پیماید و تاریخچه ساخت (لایه‌ها، حجم، سن و دستور مربوطه) را چاپ می‌کند (مشابه دستور docker history).
- **remove_image_ref:** فایل متادیتای یک ایمیج را حذف می‌کند که به منزله پاک‌کردن تگ آن ایمیج است (اما لایه‌ها را بلافاصله پاک نمی‌کند).

۲-۳-۲۰ توابع مجموعه رشته (String Set)

لایه‌هایی که در فرآیند ساخت ایمیج ایجاد شده‌اند، اما در هیچ ایمیج نهایی‌ای استفاده نشده‌اند، باید پاکسازی شوند تا فضای دیسک هدر نرود. برای ردیابی لایه‌های در حال استفاده از ساختار str_set

استفاده شده است:

- **str_set_free:** حافظه اختصاص یافته به مجموعه را آزاد می‌کند.
- **str_set_contains:** بررسی می‌کند آیا یک رشته (شناسه لایه) در مجموعه وجود دارد یا خیر.
- **str_set_add:** به صورت پویا آرایه را تغییر اندازه داده و یک کپی از رشته جدید به مجموعه اضافه می‌کند.

۲-۳-۲۱ توابع جستجو و نشانه‌گذاری

- **mark_layer_chain_used:** شناسه یک لایه را گرفته و با پیمایش والدها (parent) به سمت پایین، تمام لایه‌های موجود در این زنجیره را به مجموعه لایه‌های "در حال استفاده" (used set) اضافه می‌کند.
- **collect_used_layers:** روی تمام فایل‌های متادیتای ایمیج‌ها پیمایش کرده و زنجیره تمام ایمیج‌های موجود را از طریق تابع قبل، به عنوان لایه‌های مستعمل نشانه‌گذاری می‌کند.

۲-۳-۲۲ توابع پاکسازی (Pruning)

- **cleanup_cache_entries:** پوشه کش را بررسی کرده و اگر فایل کشی به لایه‌ای اشاره کند که حذف شده و وجود ندارد، آن فایل کش را حذف می‌کند (جلوگیری از نشانگرهای مرده).
- **prune_unused_layers:** فرآیند نهایی GC را در یک حلقه تکرارشونده اجرا می‌کند:

۱. با فراخوانی collect_used_layers، لیست تمامی لایه‌های مورد نیاز ایمیج‌ها را می‌سازد.

۲. دایرکتوری لایه‌ها را می‌پیماید. اگر پوشه‌ای در لیست مستعمل‌ها نباشد، آن را به همراه تمام محتویاتش (remove_recursive) حذف می‌کند.

۳. کش‌ها را با cleanup_cache_entries به‌روزرسانی می‌کند.

۴. این فرآیند را تکرار می‌کند تا زمانی که هیچ لایه‌ای در یک دور حذف نشود (برای اطمینان از حذف کامل لایه‌های یتیم شده متوالی).

۲-۴ تحلیل و توضیح فایل config.c

در نسخه اولیه فایل config.c، تابع validate_config تنها برای دو زیردستور run و exec طراحی شده بود و صرفاً وجود نام کانتینر، دستور اجرا و base image/base dir را بررسی می‌کرد.

در نسخه نهایی، با گسترش قابلیت‌های پروژه (مانند build, history, images, rmi, prune)، منطق اعتبارسنجی نیز بازطراحی شده و به‌صورت وابسته به نوع زیردستور (subcommand) پیاده‌سازی شده است.

۲-۴-۱ گسترش لیست Subcommandها

در نسخه جدید، مقدار subcommand می‌تواند یکی از حالت‌های زیر باشد:

- RUN
- BUILD
- HISTORY
- IMAGES
- RMI
- PRUNE

در صورت مشخص نبودن زیردستور، پیام خطا شامل تمام گزینه‌های معتبر نمایش داده می‌شود.

۲-۴-۲ تفکیک اعتبارسنجی بر اساس نوع دستور

برخلاف نسخه اولیه که همه بررسی‌ها به‌صورت یکپارچه انجام می‌شد، در نسخه جدید برای هر زیردستور بلوک اعتبارسنجی جداگانه تعریف شده است:

- برای RUN: بررسی نام کانتینر، دستور اجرایی و وجود base image یا base dir.
- برای BUILD: بررسی وجود مسیر Zockerfile (گزینه f-) و تگ ایمیج (گزینه t-).
- برای HISTORY و RMI: بررسی وجود مرجع ایمیج.

این ساختار باعث شده اعتبارسنجی دقیقاً مطابق نیاز هر دستور انجام شود.

۲-۴-۳ سازگاری با دستورات جدید پروژه

با اضافه شدن قابلیت build و مدیریت ایمیج‌ها، فیلدهای جدیدی مانند zockerfile و image_ref در فرآیند اعتبارسنجی وارد شده‌اند.

در نتیجه، فایل config.c از یک اعتبارسنج ساده برای اجرای کانتینر، به یک سیستم اعتبارسنجی چنددستوره برای کل ابزار Zocker توسعه یافته است.

۲-۵ تحلیل و توضیح فایل `setup.c`

در نسخه جدید فایل setup.c ساختار اولیه که صرفاً یک OverlayFS mount ساده انجام می‌داد، گسترش داده شده و چند قابلیت جدید به آن اضافه شده است.

۲-۵-۱ اضافه شدن تابع ensure_dir

در نسخه قبلی ایجاد دایرکتوری‌ها مستقیماً با mkdir انجام می‌شد. در نسخه جدید تابع ensure_dir اضافه شده که ابتدا وجود مسیر را بررسی می‌کند و در صورت نیاز آن را ایجاد می‌کند. همچنین اگر مسیر وجود داشته ولی دایرکتوری نباشد، خطا برمی‌گرداند. این تغییر باعث مدیریت ایمن‌تر ساختار فایل‌سیستم می‌شود.

۲-۵-۲ گسترش تابع setup_zocker_dir

در نسخه قبلی تنها یک مسیر اصلی ایجاد می‌شد، اما در نسخه جدید این تابع تمام زیرشاخه‌های موردنیاز پروژه شامل:

- containers
- layers
- layer-links
- images
- cache

- build-tmp

را ایجاد می‌کند. این تغییر برای پشتیبانی از معماری لایه‌ای و مکانیزم caching اضافه شده است.

۲-۵-۳ اضافه شدن مکانیزم Resolve کردن Base Image

در نسخه قبلی، base image صرفاً به‌صورت یک مسیر فایل در نظر گرفته می‌شد. در نسخه جدید تابع resolve_base_chain اضافه شده که می‌تواند:

- یک مسیر مستقیم روی سیستم فایل،
- یک ایمیج ذخیره‌شده در Zocker
- یا یک ایمیج موجود در Docker

را دریافت کرده و زنجیره کامل lowerdir را تولید کند.

برای ایمیج‌های Docker از دستور docker inspect جهت استخراج UpperDir استفاده می‌شود و سپس زنجیره لایه‌ها بازسازی می‌گردد.

۲-۵-۴ بازسازی زنجیره لایه‌های Docker

تابع build_docker_chain_from_upper اضافه شده است. این تابع با خواندن فایل lower در ساختار داخلی OverlayFS داکر، تمام لایه‌های قبلی را استخراج کرده و به‌صورت یک رشته lowerdir قابل استفاده در mount تولید می‌کند.

۲-۵-۵ اضافه شدن اعتبارسنجی مسیرهای Overlay

تابع validate_overlay_paths برای جلوگیری از پیکربندی نادرست OverlayFS اضافه شده است. در این تابع بررسی می‌شود که مسیرهای upperdir و workdir داخل یکی از lowerdir ها قرار نگرفته باشند. این بررسی با استفاده از realpath انجام می‌شود.

۲-۵-۶ تغییر در امضای setup_container_dir

در نسخه جدید:

- اندازه بافرها افزایش یافته است.
- مسیر container root جدا از merged نگهداری می‌شود.
- ابتدا base chain resolve می‌شود و سپس mount انجام می‌گیرد.

این تغییرات باعث شده تابع بتواند با image chain های چندلایه و مکانیزم build cache سازگار باشد.

۲-۶ تحلیل و توضیح فایل build.c

۲-۶-۱ ثابت‌ها و هدرها

در ابتدای فایل، هدرهای استاندارد لینوکس برای مدیریت فرآیندها (sys/wait.h)، فایل‌سیستم (sys/mount.h، sys/stat.h) و مدیریت رشته‌ها فراخوانی شده‌اند. همچنین ثابت‌های زیر تعریف شده‌اند:

- MAX_STAGES: حداکثر تعداد مراحل (stages) در یک بیلد (۳۲).
- MAX_LOCAL_ARGS: حداکثر تعداد آرگومان‌های محلی (ARG) که می‌توان ذخیره کرد (۱۲۸).

۲-۶-۲ ساختار struct arg_map

این ساختار برای ذخیره‌سازی آرگومان‌های بیلد (Key-Value pairs) استفاده می‌شود.

- items: آرایه‌ای از build_arg برای نگهداری کلیدها و مقادیر.
- count: تعداد آرگومان‌های ذخیره شده فعلی.

۲-۶-۳ ساختار struct stage_ctx

این ساختار وضعیت یک مرحله (Stage) از فرآیند بیلد را نگهداری می‌کند:

- name: نام مرحله (مثلاً builder یا شماره ایندکس).
- base_chain: مسیر زنجیره لایه‌های پایه (مسیری که ایمیج پایه در آن قرار دارد).
- top_layer: شناسه (ID) بالاترین لایه ساخته شده در این مرحله.
- state_hash: هش وضعیت فعلی مرحله (برای استفاده در مکانیزم Caching).
- workdir: دایرکتوری کاری فعلی در داخل کانتینر.
- args: آرگومان‌های اختصاصی این مرحله.
- cmd: دستور پیش‌فرض (CMD) که در نهایت روی ایمیج ست می‌شود.

۲-۶-۴ تابع arg_map_set

این تابع وظیفه افزودن یا به‌روزرسانی یک آرگومان در arg_map را دارد.

- ابتدا بررسی می‌کند که ورودی‌ها NULL نباشند.
- در حلقه تکرار، اگر کلید (key) از قبل وجود داشته باشد، مقدار (value) آن را به‌روزرسانی کرده و بازمی‌گردد.
- اگر کلید جدید باشد و ظرفیت (MAX_LOCAL_ARGS) پر نشده باشد، یک آیتم جدید به آرایه اضافه کرده و شمارنده count را افزایش می‌دهد.

۲-۶-۵ تابع arg_map_get

این تابع جستجو در لیست آرگومان‌ها را انجام می‌دهد.

- یک کلید دریافت کرده و در map جستجو می‌کند.
- اگر کلید پیدا شد، اشاره‌گر به مقدار (value) را برمی‌گرداند؛ در غیر این صورت NULL بازمی‌گرداند.

۲-۶-۶ تابع arg_map_copy

وظیفه کپی عمیق (Deep Copy) یک نقشه آرگومان به دیگری را دارد. ابتدا حافظه مقصد را با memset صفر کرده و سپس محتویات را با memcpy کپی می‌کند.

۲-۶-۷ تابع init_cli_args_map

این تابع آرگومان‌هایی که از طریق خط فرمان (CLI) به برنامه داده شده‌اند (در ساختار config) را به فرمت داخلی arg_map تبدیل می‌کند.

۲-۶-۸ تابع substitute_args

یکی از توابع حیاتی برای تفسیر متغیرها در Dockerfile است. این تابع یک رشته ورودی را گرفته و متغیرهایی به فرم $VAR یا ${VAR} را با مقادیر موجود جایگزین می‌کند.

- رشته را کاراکتر به کاراکتر پیمایش می‌کند.
- با رسیدن به علامت $، بررسی می‌کند که آیا فرمت $$ (برای فرار از کاراکتر) است یا شروع یک متغیر.
- نام متغیر را استخراج می‌کند (با پشتیبانی از آکولاد یا بدون آن).
- با استفاده از arg_map_get مقدار متغیر را پیدا کرده و در بافر خروجی جایگزین می‌کند.
- اگر متغیر پیدا نشود، با رشته خالی جایگزین می‌گردد.

۲-۶-۹ تابع make_temp_dir

یک دایرکتوری موقت منحصربه‌فرد برای عملیات بیلد ایجاد می‌کند.

- از UUID و PID (شناسه فرآیند) برای تضمین یکتایی نام پوشه استفاده می‌کند.
- مسیر نهایی معمولاً در ZOCKER_BUILD_TMP_DIR ساخته می‌شود.

۲-۶-۱۰ تابع compute_state_hash

برای سیستم Caching استفاده می‌شود.

- هش والد (parent_hash) را با توضیحات دستور فعلی (descriptor) ترکیب می‌کند.
- نتیجه را هش کرده (احتمالا SHA256) و در out_hash قرار می‌دهد. این هش مشخص می‌کند که آیا این وضعیت قبلاً ساخته شده است یا خیر.

۲-۶-۱۱ تابع resolve_stage_chain

مسیر فایل‌سیستم برای وضعیت فعلی را تعیین می‌کند.

- اگر مرحله دارای top_layer باشد، زنجیره لایه‌ها را از بالا به پایین حل می‌کند.
- اگر هنوز لایه‌ای ساخته نشده باشد، مسیر base_chain (ایمیج پایه) را برمی‌گرداند.

۲-۶-۱۲ تابع run_in_chroot

این تابع دستورات RUN را در محیط ایزوله اجرا می‌کند.

- **آماده‌سازی:** ابتدا مسیرهای rootfs و workdir را آماده می‌کند.
- **بررسی شل:** وجود /bin/sh در سیستم فایل ریشه را بررسی می‌کند.
- **Fork:** یک فرآیند فرزند ایجاد می‌کند.
- **Process: Child**
  - با chroot ریشه فایل‌سیستم را به rootfs تغییر می‌دهد.
  - با chdir به دایرکتوری کاری می‌رود.
  - با execl دستور را توسط /bin/sh -c اجرا می‌کند.
- **Process: Parent** منتظر اتمام فرآیند فرزند می‌ماند (waitpid) و کد خروج (Exit Status) را بررسی می‌کند.

۲-۶-۱۳ تابع basename_of

یک تابع کمکی ساده که نام فایل را از یک مسیر کامل استخراج می‌کند (شبیه به دستور basename در لینوکس).

۲-۶-۱۴ تابع copy_into_rootfs

وظیفه کپی کردن فایل‌ها یا دایرکتوری‌ها از سیستم میزبان (Host) به داخل rootfs کانتینر را دارد.

- مسیر مقصد در کانتینر را نرمال‌سازی می‌کند.
- بررسی می‌کند که مقصد یک دایرکتوری است یا فایل.
- با استفاده از copy_path_recursive عملیات کپی را انجام می‌دهد.

۲-۶-۱۵ تابع mount_overlay

این تابع حیاتی، سیستم فایل OverlayFS را مانت می‌کند.

- **بررسی امنیتی:** بررسی می‌کند که مسیرهای upper و work داخل هیچکدام از مسیرهای lower نباشند تا از خطای دوری (Circular dependency) جلوگیری شود.
- **ساخت آپشن‌ها:** رشته mount options را با فرمت lowerdir=...,upperdir=...,workdir=... می‌سازد.
- **Mount:** تابع سیستمی mount را با نوع overlay فراخوانی می‌کند.

۲-۶-۱۶ تابع with_stage_snapshot

این تابع برای دستور COPY –from=... استفاده می‌شود.

- یک اسنپ‌شات موقت از وضعیت فایل‌سیستمِ یک مرحله (Stage) دیگر ایجاد می‌کند.
- با استفاده از mount_overlay لایه‌های آن مرحله را در یک مسیر موقت مانت می‌کند تا بتوان از آن فایل کپی کرد.

۲-۶-۱۷ تابع create_layer_dirs

ساختار فیزیکی یک لایه جدید را روی دیسک ایجاد می‌کند. برای هر لایه پوشه‌های زیر ساخته می‌شوند:

- diff: تغییرات فایل‌سیستم در این لایه.
- work: دایرکتوری کاری مورد نیاز OverlayFS.
- lower: فایلی حاوی لیست لایه‌های پایین‌دستی.
- link: فایلی حاوی شناسه کوتاه شده (Short ID).
- همچنین یک symlink در پوشه layer_links ایجاد می‌کند تا دسترسی سریع‌تر به لایه فراهم شود.

۲-۶-۱۸ ساختار struct run_apply_ctx و تابع apply_run_layer

کانتکست شامل دستور و دایرکتوری کاری است. تابع apply_run_layer صرفاً تابع run_in_chroot را با مسیر merged فراخوانی می‌کند.

۲-۶-۱۹ ساختار struct workdir_apply_ctx و تابع apply_workdir_layer

وظیفه ایجاد دایرکتوری WORKDIR در لایه جدید را دارد. از ensure_dir_path استفاده می‌کند.

۲-۶-۲۰ ساختار struct copy_apply_ctx و تابع apply_copy_layer

پیچیده‌ترین تابع اعمال تغییرات است که دو حالت دارد:

۱. **کپی از مرحله دیگر (from_stage):**

- با استفاده از with_stage_snapshot مرحله مبدأ را مانت می‌کند.
- فایل مورد نظر را از آنجا به لایه جاری کپی می‌کند.
- پس از اتمام، آن‌مانت (umount) و پاکسازی می‌کند.

۲. **کپی از کانتکست بیلد:**

- فایل را مستقیماً از مسیر کانتکست (روی هاست) به داخل لایه کپی می‌کند.

۲-۶-۲۱ ساختار struct add_apply_ctx و توابع مرتبط

- **download_url_to_file:** یک پردازش فرزند ایجاد کرده و با استفاده از دستور curl، فایل را از اینترنت دانلود می‌کند.
- **apply_add_layer:** دستور ADD را مدیریت می‌کند.
  - اگر منبع یک URL باشد، ابتدا آن را در یک دایرکتوری موقت دانلود کرده و سپس به روت‌فایل‌سیستم کپی می‌کند.
  - اگر فایل محلی باشد، مانند دستور COPY رفتار می‌کند.

۲-۶-۲۲ تابع apply_noop_layer

این تابع کاری انجام نمی‌دهد و مقدار ۰ برمی‌گرداند. معمولاً برای ایجاد لایه‌های متادیتا یا final stage که تغییری در فایل‌سیستم نمی‌دهند استفاده می‌شود.

۲-۶-۲۳ تابع create_layer

این تابع برای ساختن یک لایه جدید در اثر یک دستور در Zockerfile است.
یک ورودی مهم این تابع یک function pointer به اسم apply_fn خواهد بود. این تابع بر اساس نوع دستور به تابع ورودی داده می شود.
مثلا اگر دستور copy باشد، تابع apply_copy_layer به این تابع پاس داده خواهد شد و اگر دستور add باشد apply_add_layer.
ورودی مهم دیگر، یک ورودی *void به نام apply_ctx است. این ورودی، از نوع struct ای خواهد بود که برای دستور مد نظر ساخته‌ایم. مثلا اگر دستور copy باشد، این ورودی، اشاره‌گری خواهد بود به یک struct copy_apply_ctx که اطلاعات لازم برای همان apply_fn را دربر خواهد داشت و به عنوان ورودی آن داده خواهد شد.
یک struct stage_ctx هم برای داشتن اطلاعات stage کنونی به منظور محاسبه hash لایه جدید و ... به تابع ورودی داده می‌شود.

بعلاوه descriptor و متن دستور مدنظر هم ورودی‌های این تابع هستند.
با descriptor های دستورات مختلف در تابع build_image_from_config آشنا می‌شوید.
این تابع مراحل زیر را طی می‌کند:

- ابتدا hash لایه جدید را با استفاده از hash وضعیت قبلی که در stage آمده است و descriptor دستور جدید محاسبه می‌کند.
- سپس به دنبال این hash در پوشه cache می‌گردد و اگر پیدا کرد، صرفا لایه بالایی stage را برابر شناسه لایه پیدا شده قرار می‌دهد و hash را در stage با hash جدید محاسبه شده جایگزین می‌کند و اجرای تابع در این حاللت به پایان می‌رسد.
- اگر لایه پیدا نشد، باید آن را بسازیم. بنابرین، اولین کاری که می‌کنیم این است که با استفاده از لایه بالایی وضعیت کنونی که در stage داریم، زنجیزه overlay تا پدر این لایه را محاسبه می‌کنیم.
- سپس با استفاده از create_layer_dirs که توضیح دادیم، پوشه‌های work و diff و فایل‌های lower و link را می‌سازیم و با زنجیره overlay که بدست آوردیم توی lower, link را پر می‌کنیم (این کار هم در همان تابع انجام می‌شود)
- حال باید برای اجرای تابع apply_fn یک کانتینر موقت بسازیم. بنابراین، یک پوشه موقت با پیشوند build می‌سازیم و توی آن یک پوشه merged می‌سازیم.
  از زنجیره overlay که محاسبه کرده بودیم به عنوان زنجیره lower، از پوشه diff به عنوان upper از پوشه work توی لایه به عنوان work و از پوشه merged توی پوشه موقت به عنوان merged استفاده می‌کنیم و merged را به عنوان mount point قرار می‌دهیم تا یک کانتینر موقت ایجاد شود و با همان root تابع apply_fn را اجرا می‌کنیم.
- بعد از اجرای تابع apply_fn پوشه merged را unmount کرده و کل پوشه موقت را حذف می‌کنیم.
- در نهایت هم نوبت به ساختن و ذخیره کردن فایل meta می‌شود که بر اساس داده‌هایی که داریم و با استفاده از تابع write_layer_metadata که در image_store.c دیدیم، این کار را انجام می‌دهیم.

۲-۶-۲۴ توابع مربوط به پارس کردن Zockerfile

- تابع parse_two_tokens:

این تابع صرفا دو token اول رشته ورودی (tokenize شده با white space) را در محل‌هایی که دو اشاره‌گر به آن‌ها اشاره می‌کنند،ذخیره می‌کنند.

- تابع parse_copy_tokens:

این تابع مخصوص دستور copy نوشته شده‌است. بر اساس اینکه در رشته ورودی form–داریم یا خیر (که اگر multi-stage building باشد داریم) اسم stage ای که from به آن اشاره دارد (در صورت وجود،این مورد یا عدد خواهد بود و یا alias ای که جلوی AS می‌نویسیم) ، مبداء و مقصد کپی را استخراج می‌کند و در متغیرهایی که اشاره‌گر به آن‌ها ورودی تابع است ذخیره می‌کند.

- تابع parse_base_and_alias:

این مورد برای multi-stage building و زمانی که می‌نویسیم FROM base-img-ref AS alias استفاده می‌شود و base-img-ref و alias را از رشته ورودی استخراج می‌کند. (البته اگر کلمه AS در ورودی نباشد، صرفا token اول را به عنوان base درنظر می‌گیرد.)

- تابع stage_index_by_name:

در from در دستور copy می‌نوانیم هم alias یک استیج را بیاوریم و یا شماره index آن استیج در آرایه stage ها (که جلوتر در تابع build_image_from_container با این آرایه مواجه خواهید شد.)

بنابراین، این تابع بررسی می‌کند آیا تمام کاراکترهای رشته ورودی، عدد هستند یا خیر. اگر بودند، آن را به عنوان اندیس در آرایه استیج‌ها در نظر می‌گیرد و به عدد صحیح تبدیل کرده بر می‌گرداند.

اگر نبودند، آن را به عنوان alias درنظر می‌گیرد و به دنبال stage ای می‌گردد که نام آن برابر این alias باشد و اندیس همان stage را بر می‌گرداند.

- تابع parse_arg_kv:

در تعریف ARG در یک Zockerfile ممکن است برای آن مقدار پیش‌فرض تعریف کنیم و ممکن هم هست این کار را نکنیم.

اگر پیش‌فرض تعریف کنیم، به صورت ARG=defalut خواهد بود. این تابع رشته ورودی را می‌گیرد. به دنبال علامت = توی آن می‌گردد. اگر این علامت وجود نداشت،کل رشته را به

عنوان کلید آرگومان درنظر می‌گیرد و اگر وجود داشت، از آن به قبل را به عنوان کلید و از آن به بعد را به عنوان مقدار پیش‌فرض در نظر می‌گیرد و متغیر has_default را هم مقداردهی می‌کند.

۲-۶-۲۵ تابع build_image_from_config

شاید بتوان گفت این تابع مهم‌ترین تابع این پروژه است. در این تابع، از روی یک کانفیگ ورودی،یک image را به صورت لایه لایه می‌سازیم.

- ابتدا، آرگومان‌های ورودی کاربر را که در فیلد "آرگومان‌ها" ی کانفیگ هستند را به مجموعه آرگومان‌های global و همینطور مجموعه آرگومان‌های cli_args ذخیره می‌کنیم. (در ادامه منظور از آرگومان، مجموعه‌ای زوج‌های کلید، مقدار است)
- سپس از روی مسیر Zockerfile که در کانفیگ ذخیره است، Zockerfile دسترسی پیدا می‌کنیم و در یک حلقه while خط به خط شروع به خواندن آن می‌کنیم.
- white space های سر و ته خطی که خواندیم را می بریم و اگر خط خالی بود یا با نشانه کامنت شروع می‌شد، به خط بعد می‌رویم.
- در غیر این صورت، خط را تا رسیدن به اولین space می‌خوانیم و حروفش را به upperCase تبدیل می‌کنیم. این زیر رشته، دستور ما خواهد بود.
  بقیه رشته را نیز در متغیر rest نگه‌داری می‌کنیم. حالا نوبت به آن رسیده‌است که روی دستور، حالت‌بندی کنیم.
  در هر دستور در صورت نیاز لایه یا stage جدیدی می‌سازیم و مجموعه stage های ساخته شده را در یک آرایه ذخیره می‌کنیم.
- دستور ARG:
  در این صورت، ابتدا با تابع parse_arg_kv کلید آرگومان و مقدار دیفالت آن(در صورت وجود) را پیدا می کنیم .
  حتی اگر مقدار default داشت، ممکن است این مقدار دیفالت، خود شامل متغیرهایی باشد. باید آن را با تابع substitute_args به رشته‌ای تبدیل کنیم که متغیرهای درون آن با مقادیرشان جایگزین شده‌اند. اگر تا کنون stage ای ساخته شده‌بود، به تابع substitute_args می‌گوییم از میان آرگومان‌های ذخیره شده در stage کنونی (که البته، آرگومان‌های global زیرمجموعه‌ای

از آن‌ها خواهند بود) به‌دنبال آرگومان‌های توی مقدار دیفالت بگردد.
اگر تاکنون stage ای ساخته نشده بود هم باید از میان آرگومان‌های global به‌دنبال متغیرهای توی مقدار دیفالت بگردیم.
حال، اگر کاربر در دستور خود به این آرگومان مقدار داده بود (که یعنی این آرگومان در cli_args پیدا می‌شود) همان مقدار را به عنوان مقدار نهایی این آرگومان ذخیره می‌کنیم. در غیر این صورت، اگر مقدار دیفالت وجود داشت، مقدار دیفالت را به عنوان مقدار نهایی این آرگومان ذخیره می‌کنیم. ذر صورتی که مقدار دیفالت هم نداشت در بین آرگومان‌های stage کنونی (اگر stage ای ساخته نشده بود، آرگومان‌های globa) به دنبال کلید این آرگومان می‌گردیم، شاید این آرگومان را پیش از این نیز تعریف و مقداردهی کرده باشیم.
اگر هیچ‌کدام از موارد بالا رخ نداد، مقدار نهایی آرگومان را هی می‌گذاریم.
در نهایت، در صورتی که stage ای وجود داشت، این ارگومان را در آرگومان‌های آن stage ذخیره می‌کنیم (یا اگر از قبل وجود داشت مقدار نهایی آن را آپدیت می‌کنیم) و اگر هنوز استیجی ساخته نشده‌بود، آن را در آرگومان‌های global ذخیره می‌کنیم.

- دستور FROM یا BASE_DIR:

در این حالت، باید یک stage جدید بسازیم. بنابراین اگر ظرفیت ساخت stage نداشتیم خطا بر می‌گردانیم.
در غیر این صورت، ابتدا با تابع parse_base_and_alias نام base و alias را استخراج می‌کنیم.
حال، شروع به ساختن stage جدید می‌کنیم. در ابتدا، کل آرگومان‌های global را به آرگومان‌های استیج جدید کپی می‌کنیم و workdir آن را برابر rootfs می‌گذاریم (که طبیعی است چون بعد از دستور from یا base_dir به یک root جدید نگاه می‌کنیم) و سپس اگر alias تهی نبود، نام stage جدید را برابر آن alias و اگر تهی بود برابر اندیس آن در آرایه stage ها می‌گذاریم.
برای base_chain استیج حدید، اگر دستور FROM بود از تابع resolve_base_chain که در setup.c دیدیم استفاده می‌کنیم و اگر دستور base_dir بود و آدرس نسبی بود ، آدرس را نسبت به مسیر Zockerfile می‌سنجیم و اگر آدرس مطلق بود همان آدرس را به عنوان base_chain قرار می‌دهیم.
Descriptor این دستور به صورت BASE|base_chain است که از آن برای محاسبه hash استیج استفاده می‌کنیم.

- اگر دستور ما هیچ یک از دستورات ARG, BASE_DIRT, FROM نبود ولی هنوز stage ای

ساخته نشده بود، باید خطا برگردانیم.

- دستور RUN:

در این حالت ابتدا با substitute_args دستوری که باید اجرا شود را resolve می‌کنیم.
descriptor دستور RUN به صورت RUN|wd=working_dir|cmd=command است.
یک run_ctx با استفاده از working_directory استیج کنونی و command استخراج شده و رمزگشایی شده می‌سازیم و یک لایه با استفاده از تابع create_layer که به آن تابع apply_run_layer و run_ctx ساخته شده را پاس دادیم می‌سازیم.

- دستور WORKDIR:

برای این دستور هم الگوی مشابه ساختن لایه تکرار می‌شود. فقط این‌بار مسیری که باید workdir را برابر آن بگذاریم را resolve کرده سپس با تابع normalize_container_path یک مسیر abstract و بدون . یا .. بدست می‌آوریم.
بعلاوه descriptor این دستور به فرم WORKDIR|path=normalized_path است.
پس از ساخت لایه، از آنجا که apply_workdir_layer صرفا از وجود پوشه جدید مطمئن می‌شود، باید به صورت دستی workdir را در stage کنونی عوض کنیم.

- دستور COPY:

این بار هم باید بقیه دستور را resolve کنیم (متغیرها را با مقادیر جایگزین کنیم)، با تابع parse_copy_tokens مبداء، مقصد و stage ای که from (در صورت وجود) به آن اشاره دارد را پیاد کنیم.
اگر from داشتیم، از روی اسم stage که پیدا کردیم، با استفاده از تابع stage_index_by_name اندیس stage را در میان آرایه stage ها پیدا کنیم ، Descriptor ای به فرم COPY|from=sth|src=sthelse|src_state=sthelse|dst=sthelse بسازیم و یک copy_ctx مناسب بسازیم.
در غیر این صورت، اگر مسیر source نسبی است آن را نسبت به مسیر Zockerfile بسنجیم و با استفاده از hash_path_recursive یک hash از مسیر source بسازیم. Descriptor در این حالت به فرم COPY|src=sth|src_hash=sthelse|dst=sthelse است.
در نهایت باید یک لایه با استفاده از create_layer که به آن تابع apply_copy_layer را به همراه copy_ctx ساخته شده پاس داده‌ایم بسازیم. دستور ADD:
این دستور بسیار شبیه به دستور copy است. فقط descriptor آن در حالتی که دستور حاوی url باشد به صورت ADD|url=sth|dst=sthelse و در صورتی که حاوی مسیر local باشد به

صورت ADD|src=sth|src_hash=sthelse است.

- CMD :

دز این دستور کافیست فیلد cmd استیج کنونی را آپدیت کنیم.

در نهایت و پس از خواندن ZockerFile اگر stage ای ساخته نشده بود خطا بر می‌گردانیم، چون هر Zockerfile معتبر باید حداقل یک دستور FROM یا BASE_DIR داشته باشد.
بعلاوه، با تابع ensure_final_stage_has_layer مطمئن می‌شویم ایمیج ساخته شده حداقل یک لایه داشته باشد و درنهایت با استفاده از اطلاعات stage نهایی و نام و تگ موجود در image_ref در کانفیگ، فایل metadata مربوط به ایمیج خود را می‌سازیم.

۲-۷ توضیح فایل main.c

۲-۷-۱ تابع append_run_command

این تابع وظیفه دارد token های دستور shell ای که کاربر در دستور zocker run می‌نویسد را یکی یکی و با یک space فاصله نسبت به token قبلی به فیلد command کانفیگ ورودی خود اضافه کند و هرگاه طول دستور از بیشینه مجاز طول این فیلد بیشتر شد، خطا بدهد.

۲-۷-۲ تابع parse_build_arg_value

این تابع وظیفه دارد بررسی کند آیا کانفیگ ورودی ظرفیت اضافه شدن یک آرگومان بیلد دیگر را هم دارد یا نه. اگر داشت به دنبال علامت مساوی در رشته ورودی‌اش می‌گردد و اگر پیدا کرد، قبل از آن را به‌عنوان کلید و بعد از آن را به عنوان value ی یک آرگومان جدید به لیست آرگومان‌های کانفیگ ورودی اضافه می‌کند. (در صورتی که طول کلید بیش از حد مجاز باشد یا رشته ورودی علامت = نداشته باشد، خطا بر می‌گرداند.)

۲-۷-۳ حالت‌بندی روی token های دستور کاربر

این بخش همانند تمرین هشت است، فقط چند حالت به آن اضافه شده‌است.
حالت‌های اضافه شده، token های build, history, images, rmi, prune هستند که دستورات جدید پروژه نسبت به تمرین هشت هستند و با دیدن آن‌ها فیلد subcommand را در کانفیگ تعیین می‌کنیم.

همچنین کاربر می‌تواند با f- یا file- مسیر Zockerfile مدنظرش را بدهد. توجه کنید آدرس Zockerfile باید به صورت abstract باشد.
با t- هم می‌توان نام و تگ ایمیج نهایی را مشخص کرد و با build-arg– هم می‌توان به آرگومان‌های تعریف شده در Zockerfile مقدار دهی کرد. (مقداردهی باید به صورت KEY=VALUE باشد تا تابع parse_build_arg_value روی آن کار کند.)

۲-۷-۴ حالت بندی روی نوع دستور

- دستور RUN: در این حالت، ابتدا یک کانتینر از روی کانفیگ می‌سازیم (با کپی کردن فیلدهای آن در فیلدهای کانتینر) و بعد تابع run_container که در run.c است و مشابه همان چیزی است که در تمرین هشتم داشتیم را صدا می‌زنیم.
- دستور BUILD: در این حالت، تابع build_image_from_config که در توضیح فایل build.c آن را توضیح دادیم را صدا می‌زنیم.
- دستور HISTORY: در این صورت تابع print_image_history در فایل image_store.c را صدا می‌زنیم.
- دستور IMAGES: در این صورت تابع list_images در image_store.c را صدا می‌زنیم.
- دستور RMI: در این صورت تابع remove_image_ref در image_store.c را صدا می‌زنیم.
- PRUNE: در این صورت تابع prune_unused_layers در image_store.c را صدا می‌زنیم.

۳ تست

برای تست کردن کد، ابتدا یک فایل script که به پیوست ارسال می‌شود را اجرا کردیم و سپس چند تست دستی انجام دادیم. در اجرای اسکریپت مراحل زیر طی شد:

۳-۱ مراحل پیش‌نیاز و آماده‌سازی محیط

اسکریپت پیش از شروع تست‌های اصلی، عملیات زیر را برای اطمینان از سلامت محیط اجرا انجام می‌دهد:

- **بررسی سطح دسترسی:** از آنجایی که عملیات mount و chroot نیاز به دسترسی ریشه دارند، اسکریپت ابتدا EUID کاربر را چک می‌کند.
- **بررسی پشتیبانی از OverlayFS:** با استفاده از تابع preflight_overlay، یک مانت آزمایشی انجام می‌شود تا اطمینان حاصل شود که هسته لینوکس از سیستم‌فایل لایه‌ای پشتیبانی می‌کند.
- **ایجاد محیط ایزوله:** دایرکتوری‌های موقتی در /tmp برای ذخیره‌سازی داده‌ها (STORE_DIR) و گزارش‌ها ایجاد می‌شوند تا تست بر روی فایل‌های سیستم میزبان اثری نگذارد.

۳-۲ آماده‌سازی RootFS مینیمال

یکی از بخش‌های فنی مهم این تست، تابع copy_bin_with_libs است که یک پوشه را برای BASEDIR شدن آماده می‌کند. این تابع وظایف زیر را بر عهده دارد:

۱. کپی کردن فایل‌های اجرایی (مانند sh و cat) به داخل مسیر chroot.

۲. شناسایی وابستگی‌های کتابخانه‌ای (.so) با استفاده از دستور ldd.

۳. کپی کردن مفسر (Interpreter) و کتابخانه‌های مورد نیاز به مسیر متناظر در دایرکتوری هدف برای اطمینان از اجرای صحیح دستورات در محیط ایزوله.

۳-۳ سناریوهای آزمون

۳-۳-۱ آزمون اول: بررسی مکانیزم Cache و لایه‌بندی

در این مرحله، یک Zockerfile ساده شامل دستورات COPY و RUN ایجاد می‌شود:

- **ساخت اول:** ایمیج برای بار اول ساخته می‌شود و زمان اجرای آن ثبت می‌گردد.
- **ساخت دوم:** همان فایل دوباره بیلد می‌شود. اسکریپت بررسی می‌کند که آیا عبارت [CACHE HIT] در خروجی ظاهر می‌شود یا خیر و باز هم زمان اجرا را ثبت می‌کند.
- **تأیید صحت:** اسکریپت با مقایسه هش لایه‌ها در تاریخچه (history)، اطمینان حاصل می‌کند که لایه‌های تکراری دقیقاً بازاستفاده شده‌اند. بعلاوه زمان‌های اجرا را چاپ می‌کند تا با مقایسه آن‌ها، ببینیم build دوم چقدر کمتر طول کشیده‌است.

بخشی از کد این تست:

```
log "Simple build #1"
out1="$TEST_ROOT/build1.log"
start1=$(date +%s%N)
"$BIN" build -f "$CTX_SIMPLE/Zockerfile" -t "$IMAGE_SIMPLE_V1" | tee "$out1"
end1=$(date +%s%N)
ms1=$(( (end1 - start1) / 1000000 ))

log "Simple build #2 (same inputs, different tag -> should reuse cache)"
out2="$TEST_ROOT/build2.log"
start2=$(date +%s%N)
"$BIN" build -f "$CTX_SIMPLE/Zockerfile" -t "$IMAGE_SIMPLE_V2" | tee "$out2"
end2=$(date +%s%N)
ms2=$(( (end2 - start2) / 1000000 ))

if ! grep -q "\[CACHE HIT\]" "$out2"; then
  fail "Second build has no cache hit marker"
fi

layer1=$("$BIN" history "$IMAGE_SIMPLE_V1" | awk 'NR==2 {print $1}')
layer2=$("$BIN" history "$IMAGE_SIMPLE_V2" | awk 'NR==2 {print $1}')

if [[ -z "$layer1" || -z "$layer2" ]]; then
  fail "Could not parse top layers from history"
fi

if [[ "$layer1" != "$layer2" ]]; then
  fail "Top layer differs between identical builds (cache mismatch): $layer1 vs $layer2"
fi

echo "[PASS] Cache hit detected and top layer reused"
echo "[INFO] Build#1=${ms1}ms, Build#2=${ms2}ms"
```

۳-۳-۲ آزمون دوم: ساخت چندمرحله‌ای (Multi-stage Build)

- این تست یک استیج میانی با نام deps ایجاد شده و فایلی را تولید می‌کند.
- در استیج نهایی، با استفاده از دستور `COPY --from=deps` ، فایل تولید شده به محیط جدید منتقل می‌شود.
- در نهایت، با اجرای کانتینر (zocker run) ، وجود فایل نهایی و صحت محتوای آن تأیید می‌گردد.

بخشی از کد این تست:

```
        cat > "$CTX_MULTI/payload.txt" <<ZEOF
$PAYLOAD_MULTI
ZEOF

cat > "$CTX_MULTI/Zockerfile" <<ZEOF
BASEDIR $BASE_DIR AS deps
WORKDIR /tmp/zocker-multi-$RUN_ID
COPY payload.txt /tmp/zocker-multi-$RUN_ID/in.txt
RUN /bin/sh -c 'cat /tmp/zocker-multi-$RUN_ID/in.txt > /tmp/zocker-multi-$RUN_ID/artifact.txt'

BASEDIR $BASE_DIR
COPY --from=deps /tmp/zocker-multi-$RUN_ID/artifact.txt /tmp/zocker-multi-$RUN_ID/final.txt
CMD cat /tmp/zocker-multi-$RUN_ID/final.txt
ZEOF

log "Multi-stage build"
"$BIN" build -f "$CTX_MULTI/Zockerfile" -t "$IMAGE_MULTI" | tee "$TEST_ROOT/multi_build.log"
```

نتیجه مقایسه زمانی ساخت دو ایمیج در آزمون اول را در تصویر زیر می‌بینید. همانطور که می‌بینید اولین ساخت 38ms طول کشیده‌است، در حالیکه دومین ساخت تنها 6ms طول کشیده‌است:

```
[PASS] Cache hit detected and top layer reused
[INFO] Build#1=38ms, Build#2=6ms
```

و در تصویر زیر نیز پیام موفقیت‌آمیز بودن تست را می‌بینید:

```
[TEST] Done
[DONE] All logic checks passed
[INFO] Binary: ./zocker
[INFO] Store: /tmp/zocker_store_1771206777
[INFO] Logs: /tmp/zocker_e2e_1771206777
rahmanighadirnia@rahmanighadirnia-IdeaPad-5-15ITL05:~/OS-Project14041$
```

۴-۳ تست‌های دستی

ابتدا دستور zocker images را تست کردیم:

```
rahmanighadirnia@rahmanighadirnia-IdeaPad-5-15ITL05:~/OS-Project14041$ ./zocker images
IMAGE                TOP_LAYER                                CREATED
cache-demo:1771206777-v1        901758fc-f874-4692-bc16-cd30bf778d32    1771206779
cache-demo:1771206777-v2        901758fc-f874-4692-bc16-cd30bf778d32    1771206779
multi-demo:1771206777    32a8d871-0fa2-428f-8a43-0fe2f9d778e4    1771206781
rahmanighadirnia@rahmanighadirnia-IdeaPad-5-15ITL05:~/OS-Project14041$
```

سپس دستور zocker history را برای یکی از image ها تست کردیم:

```
rahmanighadirnia@rahmanighadirnia-IdeaPad-5-15ITL05:~/OS-Project14041$ ./zocker history cache-demo:1771206777-v1
LAYER          SIZE    AGE     INSTRUCTION
901758fc-f874-4692-bc16-cd30bf778d32    23      19m     RUN /bin/sh -c 'cat /tmp/zocker-simple-1771206777/payload.txt > /tmp/zocker-simple-1771206777/out.txt'
673c97e4-a994-4637-a976-ec2101e0d885    23      19m     COPY payload.txt payload.txt
e71fe2ca-b0a0-41b3-a648-59d2a401ec95    0       19m     WORKDIR /tmp/zocker-simple-1771206777
rahmanighadirnia@rahmanighadirnia-IdeaPad-5-15ITL05:~/OS-Project14041$
```

حال یک Zockerfile ایجاد کردیم که دستورات FROM و ARG را هم داشته باشد:

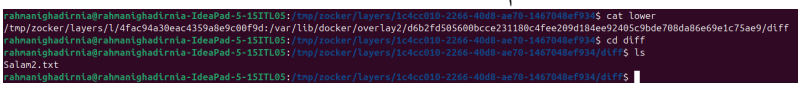
```
FROM alpine:3.22.2
ARG FileName
RUN echo "Hello from Zocker1!" > /${FileName}1.txt
RUN echo "Hello from Zocker2!" > /${FileName}2.txt
```

و در نهایت، دستور build را به این صورت صدا می‌زنیم

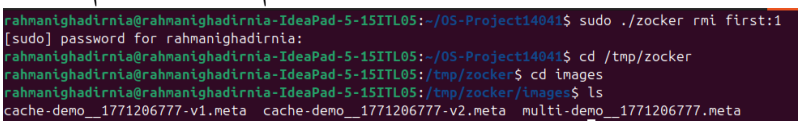
```
sudo ./zocker build -t first:1 -f
    /home/rahmanighadirnia/OS-Project14041/Zockerfile.txt --build-arg FileName=Salam
```

در تصویر زیر ساخت موفق این image را می‌بینید:

و در تصویر زیر می‌بینید که در پوشه diff بالاترین لایه این image فایل Salam2.txt وجود دارد، یعنی مقداردهی به آرگومان درست انجام شده‌است.



حالا دستور zocker rmi را برای ایمیجی که تازه ساختیم تست می‌کنیم:



همانطور که در تصویر می بینید، در لیست ایمیج‌ها، دیگر فایل meta مربوط به first:1 وجود ندارد. از آنجا که یکی از ایمیج‌ها به صورت multi-stage ساخته شده‌است و در stage میانی آن یک لایه برای هر کدام از دستورات WORKDIR, COPY, RUN ایجاد شده و دیگر استفاده نمی‌شوند، بعلاوه دو لایه برای first:1 که حذف شد ایجاد شده بود، دستور zocker prune باید 5 لایه را حذف کند که همانطور که می‌بینید این اتفاق افتاده است: