به نام خدا

داکیومنت پروژه سیستم عامل

نام اعضا: هومن هراتی، عماد یدالله پور حقیقی

1. معرفی کلی پروژه

در این پروژه یک سرویس ذخیره‌سازی محتوا محور (Content-Addressable Storage) پیاده‌سازی شده است که فایل‌ها را به قطعات (Chunk) تقسیم می‌کند، هر قطعه را بر اساس هش ذخیره می‌کند و سپس برای هر فایل یک Manifest تولید می‌کند. شناسه نهایی فایل (CID) نیز از روی مانیفست ساخته می‌شود.

معماری پروژه دو بخش دارد:

- **GATEWAY SERVER(main.py):** یک سرور HTTP که درخواست‌های uploadو downloadرا از کاربر می‌گیرد و آن‌ها را به موتور C منتقل می‌کند که توسط خودتان پیاده سازی شده بود.
- **ENGINE(c_engine.c):** هسته اصلی ذخیره‌سازی/بازیابی که روی Unix Domain Socket با main.py صحبت می‌کند و عملیاتchunking ، هش، ذخیره و assemble را انجام می‌دهد. تمام منطق اصلی در c_engine.c پیاده‌سازی شده است.

2. ساختار پوشه‌ها و فایل‌های خروجی

Engine داده‌ها را در پوشه‌ی store نگه می‌دارد. store/blocks شامل فایل‌های باینری chunk ها می‌باشد . هر chunk با هش خودش ذخیره می‌شود. store/manifest/<CID>.json شامل مانیفست فایل‌ها می‌باشد. مانیفست شامل نسخه، الگوریتم هش، chunk_size، total_size، filename و لیست chunks است.

3. نحوه اجرا

برای کامپایل اینجین c دستور زیر را وارد میکنیم:

```
gcc -O2 -pthread -Wall -Wextra -o c_engine c_engine.c
```

برای اجرا دستور زیر را وارد میکنیم:

```
./c_engine /tmp/cengine.sock
```

برای اجرای Gateway دستور زیر را وارد میکنیم:

```
python3 main.py
```

برای تست دانلود/آپلود دستورات زیر را وارد میکنیم:

```
CID=$(curl -s -X POST "http://127.0.0.1:9000/upload" \
  -H "X-Filename: hosts" \
  --data-binary "@/etc/hosts" | python3 -c 'import sys,json; print(json.load(sys.stdin)["cid"])')

curl -s "http://127.0.0.1:9000/download?cid=$CID" -o out.bin
diff -q /etc/hosts out.bin && echo "OK"
```

4. ارتباط بین c_engine.c و main.py

ارتباط بین main.py و c_engine.c از طریق Unix Domain Socket و یک پروتکل فریم‌بندی باینری پیاده‌سازی شده است. Engine روی مسیری که هنگام اجرا داده می‌شود (مثلا /tmp/cengine.sock/) bind و listen می‌کند. main.py به همین سوکت connect می‌شود و پیام‌ها را با فرمت مشخص ارسال می‌کند. هر پیام از Gateway به Engine و بالعکس شامل این ساختار است:

- opcode(1 Byte)
- length(4 Byte, big-endian)
- payload(As much as length)

پس ساختار کلی به شکل زیر است:

```
[ opcode:1 ][ length:4 big-endian ][ payload:length ]
```

این فریمینگ باعث می‌شود پیام‌ها در یک stream (سوکت) قابل تفکیک باشند و Engine دقیقاً بداند چه مقدار داده باید برای هر پیام بخواند.

در Upload ، main.py مراحل زیر را انجام می‌دهد:

1. ارسال UPLOAD_START با payload شامل متادیتا (نام فایل و اندازه).
2. ارسال چند پیام UPLOAD_CHUNK که بدنه فایل را به تکه‌های باینری منتقل می‌کند.
3. ارسال UPLOAD_ENDبرای اعلام پایان آپلود.

Engineدر handle_upload دقیقاً همین ترتیب را می‌خواند:

- بعد از دریافتSTART ، نام فایل و total_size را ذخیره می‌کند.
- با دریافت هر CHUNK، همان bytes را روی یک بافر streaming می‌گیرد و بهchunk های 256KB تقسیم می‌کند.
- برای هر chunk هش محاسبه می‌شود و اگر قبلاً ذخیره شده باشد دوباره نوشته نمی‌شود.
- در پایان، مانیفست ساخته می‌شود و CID از هش مانیفست تولید می‌شود.
- پاسخ نهایی Engine به Gateway شامل JSON یا داده‌ی لازم برای برگرداندن cid است.

در Download، main.py مراحل زیر را انجام میدهد:

1. پیام DOWNLOAD_START را با payload شامل cid ارسال می‌کند.

Engine در handle_download:

- مانیفست مربوط به cid را از store/manifests/<cid>.json می‌خواند.
- سپس chunk ها را طبق index به ترتیبassemble می‌کند.
- داده خروجی به صورت چند فریم ارسال می‌شود تا Gateway بتواند پاسخ HTTP را بدون Content-Length برگرداند.

اگر Engine به هر دلیل (manifest missing ، block missing، یا mismatch) نتواند دانلود را کامل کند، یک پاسخ خطا به Gateway ارسال می‌کند تا Gateway HTTP error برگرداند. برای جلوگیری از باقی‌ماندن سوکت روی سیستم، هنگام توقف Engineمسیر سوکت unlink می‌شود. برای اینکه توقف با Ctrl+C درست کار کند، handling سیگنال به شکل اصولی پیاده‌سازی شده تا ()accept آزاد شود و برنامه به مسیر cleanup برسد.

5. منطق Chunking و ذخیره‌سازی

فایل به صورت streaming خوانده می‌شود؛ به محض پر شدن 256KB یک chunk ساخته می‌شود. chunk آخر ممکن است کوچک‌تر باشد. هر chunk هش می‌شود و با همان هش روی دیسک ذخیره می‌شود. اگر فایل‌های مختلف chunk های یکسان داشته باشند، آن chunk فقط یک بار روی دیسک وجود خواهد داشت. پس از پایان upload ، لیست chunks و متادیتا در قالب JSON نوشته می‌شود. CID از روی محتوای مانیفست ساخته می‌شود(هش متن JSON مانیفست) تا پایدار و قابل بازتولید باشد.

6. همزمانی(Concurrency)

Engine برای پشتیبانی از چند درخواست همزمان:

- اتصال‌های همزمان را می‌پذیرد(multi-client) .
- عملیات‌های سنگین مثل هش/نوشتن chunk ها در thread pool انجام می‌شود.
- هنگام خواندن مانیفست و assemble دانلود، ترتیب خروجی حفظ می‌شود.

7. رابط گرافیکی(GUI)

رابط گرافیکی خارج از ترمینال نیز فراهم شده است:

a) GUI کلاینت

فایل gui_client.py امکان انتخاب فایل وupload و وارد کردن CID وdownload را از طریق یک UI ساده فراهم می‌کند.

b) GUI اینجین

فایل engine_viewer.py امکان اجرای c_engine از داخل پنجره، نمایش زنده‌ی لاگ‌ها start/stop کردن engine و gateway و نمایش آمارstore را فراهم میکند. این GUI باعث می‌شود عملکرد Engine بدون ترمینال قابل مشاهده باشد.

8. استفاده از هوش مصنوعی

برای بخش ارتباطی بین main.py و c_engine.c از یک ابزار هوش مصنوعی به‌عنوان راهنما استفاده کردیم. روند کار به این شکل بود:

- ابتدا ساختار پیام‌ها و فریمینگ (opcode + length + payload) را از کد Gateway (main.py) استخراج و با صورت پروژه تطبیق دادیم.

- سپس توابع خواندن/نوشتن دقیق فریم‌ها (read_exact / write_exact) را در Engine طراحی کردیم تا با stream سوکت سازگار باشد.
- در نهایت جریان Upload و Download را مرحله‌به‌مرحله مطابق ترتیبی که Gateway ارسال می‌کند در Engine پیاده‌سازی کردیم و با تست صحت آن را بررسی کردیم.

سایر بخش‌ها مانند chunking، ساختار store و manifest، و همزمانی نیز با همین رویکرد (راهنمایی گرفتن برای طراحی، و سپس پیاده‌سازی و تست دستی) تکمیل شده‌اند.

9. تست‌های انجام شده

برای اطمینان از درستی پروژه این تست‌ها را اجرا کردیم:

- Upload/Download فایل کوچک و مقایسه با diff
- Upload فایل بزرگ‌تر از 256 KB و بررسی تعداد chunk ها و صحت assemble
- دانلود همزمان چندباره از یک CID و بررسی یکسان بودن خروجی‌ها
- بررسی وجود manifest و block ها در مسیر store
- تست توقف Engine با Ctrl+C و حذف شدن socket file