پروژه IPFS

مقدمه

در این پروژه هدف ما طراحی و پیاده‌سازی یک سامانه‌ی ذخیره‌سازی محتوامحور الهام‌گرفته از IPFS است که با تمرکز بر مفاهیم اصلی درس سیستم‌عامل توسعه داده شده است. در این سامانه، یک Gateway به‌عنوان واسط دریافت درخواست‌های آپلود و دانلود از کاربر عمل می‌کند و یک Engine مستقل مسئول پردازش داده‌ها، مدیریت همزمانی و انجام عملیات سطح پایین سیستم مانند ارتباط بین‌فرایندی، مدیریت نخ‌ها و ورودی/خروجی فایل‌هاست. ارتباط بین این دو بخش از طریق Unix Domain Socket برقرار می‌شود تا جداسازی فرایندی و IPC به‌صورت واقعی تجربه شود. فایل‌ها در زمان آپلود به چانک‌های ثابت تقسیم شده، هر چانک بر اساس محتوای خود هش می‌شود و به‌صورت محتوامحور ذخیره می‌گردد؛ سپس یک مانیفست شامل اطلاعات فایل و چانک‌ها ساخته می‌شود و شناسه‌ای یکتا (CID) از روی محتوای آن محاسبه می‌گردد. در زمان دانلود، داده‌ها با استفاده از مانیفست بازیابی شده و صحت هر چانک قبل از ارسال بررسی می‌شود. تمرکز اصلی این پروژه نه بر پیاده‌سازی کامل استانداردهای IPFS، بلکه بر درک عملی مفاهیمی مانند همزمانی با نخ‌ها، همگام‌سازی، IPC، طراحی پروتکل، و تضمین صحت و پایداری عملیات در سطح سیستم‌عامل است.

تست کیس ها و نحوه اجرای پروژه:

برقراری ارتباط:

ابتدا باید فایل c_engine.c را با دستور زیر build کنیم:

```
c_engine % clang -O2 -pthread \
  c_engine.c \
  blake3/blake3.c \
  blake3/blake3_dispatch.c \
  blake3/blake3_portable.c \
  blake3/blake3_neon.c \
  -Iblake3 \
  -o c_engine
```

سپس باید آن را اجرا کنیم:

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

سپس باید فایل main.py را در ترمینال جدید اجرا کنیم:

```
python3 main.py
```

پس از اجرای دستورات بالا، در ترمینالی که c_engine در حال اجراست، خط زیر پرینت می‌شود:

```
[ENGINE] listening on /tmp/cengine.sock
```

و در ترمینالی که main.py در حال اجرا است، خط زیر پرینت می شود:

```
HTTP gateway listening on http://127.0.0.1:9000
```

در این صورت، می‌توان اطمینان حاصل کرد که ارتباط برقرار گردیده است .

تست کیس ها:

1) اعتبارسنجی مسیر کامل Upload:

این تست بررسی می‌کند که سیستم بتواند یک فایل واقعی را از طریق API آپلود دریافت کند، آن را به چانک‌های استاندارد تقسیم کند، هر چانک را هش و ذخیره نماید، و در نهایت با ساخت مانیفست، یک CID یکتا برگرداند.
ارسال هدر `X-Filename` صحت پردازش متادیتای نام فایل را می‌سنجد و استفاده از `--data-binary` تضمین می‌کند داده بدون تغییر به Engine منتقل می‌شود.
باید یک ترمینال جدید ایجاد کنیم و دستور زیر را وارد کنیم:

```
curl -s -X POST \
  -H "X-Filename: test.txt" \
  --data-binary @c_engine/c_engine.c \
  http://127.0.0.1:9000/upload
```

در این صورت در این ترمینال، خط زیر نمایش داده می‌شود:

```
{"cid": "34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974"}%
```

پس از اجرای درخواست `upload/`، پاسخ JSON شامل فیلد `cid` نمایش داده می‌شود. این مقدار یک شناسه‌ی یکتا (بر پایه‌ی هش BLAKE3) برای محتوای آپلودشده است .
علامت % فقط پرامپت شِل ترمینال است و جزو پاسخ سرور نیست.

سپس در ترمینال های c_engine و main موارد زیر نشان داده می‌شود:

```
//c_engine output

[ENGINE] UPLOAD_START: name="test.txt"

[ENGINE] UPLOAD_FINISH

[ENGINE] UPLOAD_FINISH -> CID
34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974

//main.py output

127.0.0.1 - - [12/Dec/2025 13:17:32] "POST /upload HTTP/1.1" 200 -
```

با ارسال درخواست `POST /upload`، ابتدا Engine پیام **`UPLOAD_START`** را ثبت می‌کند که نشان‌دهنده‌ی آغاز فرآیند آپلود و دریافت نام فایل است. پس از دریافت کامل داده‌ها، پیام **`UPLOAD_FINISH`** ثبت می‌شود و Engine با ساخت مانیفست و محاسبه‌ی هش محتوا، یک **CID یکتا** تولید می‌کند که در لاگ با عبارت **`UPLOAD_FINISH -> CID`** نمایش داده می‌شود. در نهایت، Gateway (main.py) با ثبت لاگ HTTP و بازگرداندن کد وضعیت **OK 200**، موفقیت کامل عملیات آپلود را تأیید می‌کند.

2) بررسی خروجی مانیفستِ ساخته‌شده بعد از آپلود

بعد از آپلود، Engine یک فایل مانیفست با نام CID در مسیر `ipfs_store/manifests/` ذخیره می‌کند. این دستور همان فایل JSON را می‌خواند و با `jq` به‌صورت مرتب و قابل‌خواندن نمایش می‌دهد تا بتوانیم صحت اطلاعات مانیفست (مثل نسخه، الگوریتم هش، اندازه‌ی چانک‌ها و لیست چانک‌ها/هش‌ها) را بررسی کنیم.

جهت انجام این بررسی، در همان ترمینال سوم و جدیدمان، باید دستور زیر را بنویسیم:

```
cat c_engine/ipfs_store/manifests/34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974.json | jq
```

پس از این دستور، موارد زیر در همین ترمینال چاپ می شوند:

```
{
  "version": 1,
  "hash_algo": "blake3",
  "chunk_size": 262144,
  "total_size": 29186,
  "filename": "test.txt",
  "chunks": [
    {
      "index": 0,
      "size": 29186,
      "hash": "104f3e5f53e941c1c0347dbc14f7d6d9349332c55c52e0253054760acbb5db59"
    }
  ]
}
```

مانیفست، مشخصات فایل و لیست چانک‌های آن را نگه می‌دارد.

- `version`: نسخه‌ی فرمت مانیفست (برای سازگاری در آینده).
- `hash_algo`: الگوریتم هش مورد استفاده برای چانک‌ها (اینجا `blake3`).
- `chunk_size`: اندازه‌ی استاندارد چانک‌ها (۲۵۶KiB).
- `total_size`: اندازه‌ی کل فایل اصلی (بر حسب بایت).
- `filename`: نام فایل آپلودشده.
- `chunks`: آرایه‌ی چانک‌ها؛ هر آیتم شامل `index` (ترتیب چانک در فایل)، `size` (اندازه‌ی همان چانک)، و `hash` (هش محتوا) است.

این ساختار باعث می‌شود هنگام دانلود، سیستم بتواند چانک‌ها را به ترتیب درست کنار هم قرار دهد و با مقایسه‌ی هش‌ها، سلامت داده را تأیید کند.

در هنگام اجرای این دستور، در ترمینالی 1 و 2 که به ترتیب c_engine و main در حال اجرا هستند، انتظار نمی‌رود تا موردی چاپ گردد.

3) صحتِ کامل مسیر دانلود (End-to-End Download)

این دستور فایل را با استفاده از `cid` از endpoint `/download` دریافت می‌کند و خروجی باینری را داخل فایل `out.bin` ذخیره می‌کند. این تست نشان می‌دهد مسیر دانلود از طریق Gateway فعال است و داده‌ی بازسازی‌شده از سمت Engine بدون تغییر به کلاینت تحویل داده می‌شود.

جهت انجام این کار ابتدا دستور زیر را در ترمینال سوم اجرا می‌کنیم:

```
curl -s \

  "http://127.0.0.1:9000/download?
cid=34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974" \

  > out.bin
```

سپس به ترتیب، در ترمینالی که c_engine.c و main.py اجرا می‌شوند، موارد زیر نشان داده می شود:

```
//c_engine.c output

[ENGINE] DOWNLOAD_START:
cid="34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974"

//main.py output

127.0.0.1 - - [12/Dec/2025 13:46:01] "GET /download?
cid=34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974 HTTP/1.1" 200 -
```

لاگ اول نشان دهنده این است که Engine درخواست دانلود را دریافت کرده و فرآیند بازیابی را با CID مشخص آغاز کرده است (خواندن مانیفست، پیدا کردن چانک‌ها، صحت‌سنجی هش هر چانک و آماده‌سازی ارسال ترتیبی).
لاگ دوم نشان دهنده این است که Gateway درخواست `GET /download` را با موفقیت پردازش کرده و پاسخ **OK 200** به کلاینت برگردانده است؛ یعنی عملیات دانلود از دید API موفق بوده است.

4) دانلود همزمان (Concurrent Download)

این تست جهت این است که ۵ درخواست دانلود برای یک CID را به‌صورت موازی اجراکند تا مطمئن شویم Gateway و Engine می‌توانند چند دانلود همزمان را بدون خرابی/اختلال سرویس بدهند و خروجی‌ها یکسان باشند.
در این تست کیس، 5 بار `curl` را در پس‌زمینه (&) اجرا می‌کند، هر بار خروجی دانلود را در یک فایل جدا (`out_1.bin` تا `out_5.bin`) ذخیره می‌کند، و با `wait` صبر می‌کند تا همه‌ی دانلودها تمام شوند.

برای اجرای این دستور، در ترمینال سوم بخش زیر را میزنیم:

```
for i in {1..5}; do
  curl -s "http://127.0.0.1:9000/download?cid=34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974"
> out_$i.bin &
done
wait
```

پس از اجرای این دستور، در ترمینال سوم، موارد زیر چاپ خواهد شد:

```
[2] 10318
[3] 10319
[4] 10320
[5] 10321
[6] 10322
[4]    done       curl -s  > out_$i.bin
[2]    done       curl -s  > out_$i.bin
[3]    done       curl -s  > out_$i.bin
[6]  + done       curl -s  > out_$i.bin
[5]  + done       curl -s  > out_$i.bin
```

شماره‌ها مثل `[2] 10318` نشان می‌دهند هر دانلود به‌عنوان یک job پس‌زمینه شروع شده و PID مربوط به آن چیست. خطوط `done` یعنی هر کدام از دانلودها با موفقیت تمام شده‌اند.

سپس در ترمینال یک و دو شاهد چاپ شدن موارد زیر خواهیم بود:

```
\\c_engine.c output

[ENGINE] DOWNLOAD_START: cid="34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974"
[ENGINE] DOWNLOAD_START: cid="34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974"
[ENGINE] DOWNLOAD_START: cid="34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974"
[ENGINE] DOWNLOAD_START: cid="34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974"
[ENGINE] DOWNLOAD_START: cid="34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974"

\\main.py output

127.0.0.1 - - [12/Dec/2025 13:51:52] "GET /download?
cid=34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974 HTTP/1.1" 200 -

127.0.0.1 - - [12/Dec/2025 13:51:52] "GET /download?
cid=34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974 HTTP/1.1" 200 -

127.0.0.1 - - [12/Dec/2025 13:51:52] "GET /download?
cid=34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974 HTTP/1.1" 200 -

127.0.0.1 - - [12/Dec/2025 13:51:52] "GET /download?
cid=34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974 HTTP/1.1" 200 -

127.0.0.1 - - [12/Dec/2025 13:51:52] "GET /download?
cid=34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974 HTTP/1.1" 200 -
```

اولین لاگ ها یعنی Engine پنج اتصال/درخواست دانلود را دریافت کرده و برای هرکدام فرآیند دانلود را شروع کرده است (خواندن مانیفست، یافتن چانک‌ها، صحت‌سنجی و ارسال).
دومین لاگ ها یعنی Gateway پنج درخواست HTTP را با موفقیت پردازش کرده و برای هرکدام پاسخ **OK 200** داده است؛ پس از دید API همه‌ی دانلودهای همزمان موفق بوده‌اند.

در ادامه اگر در ترمینال سوم، بلاک کد زیر را بنویسیم، هیچ نتیجه ای چاپ نخواهد شد که نشان دهنده این است که mismatch نداریم.

```
for i in {1..5}; do
  cmp out_$i.bin c_engine/c_engine.c || echo "Mismatch $i"
done
```

5) آپلود یک فایل بزرگ چند چانکی (20MB)

این تست آپلود یک فایل بزرگ چند چانکی (۲۰MB) را به‌صورت end-to-end بررسی می‌کند تا مطمئن شویم مسیر Gateway → Engine → ذخیره‌سازی چانک‌ها → ساخت مانیفست → تولید CID برای داده‌های حجیم هم درست کار می‌کند. با `dd` یک جریان ۲۰ مگابایتی از داده‌ی صفر (`/dev/zero`) تولید می‌شود و از طریق pipe به `curl` داده می‌شود. `curl` این داده را با هدر `X-Filename: interrupted.bin` به endpoint `/upload` می‌فرستد.

```
dd if=/dev/zero bs=1M count=20 | \
curl -X POST \
  -H "X-Filename: interrupted.bin" \
  --data-binary @- \
  http://127.0.0.1:9000/upload
```

سپس پس از اجرای این دستور، در ترمینال سوم، نتایج زیر چاپ می‌شود:

```
20+0 records in
20+0 records out
20971520 bytes transferred in 0.009438 secs (2222030091 bytes/sec)
{"cid": "145a66cd3e64ef09ca295a93b0ae72e0a0f76446be66a9ef89254f8eb37f3f43"}%
```

سه خط اول، بیانگر گزارش دستور dd است که تایید میکند دقیقا 20MB فایل تولید و ارسال شده است.

خط آخر، نشان دهنده پاسخ موفق سرور است: CID یکتای محتوای آپلودشده. علامت % فقط پرامپت ترمینال است و جزو پاسخ نیست. پس از اجرای این دستور در ترمینال یک و دو به ترتیب موارد زیر چاپ می‌شوند:

```
\\c_engine.c output

[ENGINE] UPLOAD_START: name="interrupted.bin"
[ENGINE] UPLOAD_FINISH
[ENGINE] UPLOAD_FINISH -> CID 145a66cd3e64ef09ca295a93b0ae72e0a0f76446be66a9ef89254f8eb37f3f43

\\main.py output

127.0.0.1 - - [12/Dec/2025 14:02:57] "POST /upload HTTP/1.1" 200 -
```

در لاگ اول، Engine با ثبت پیام **UPLOAD_START** آغاز آپلود فایل بزرگ را اعلام می‌کند و نام فایل دریافتی (`interrupted.bin`) را نشان می‌دهد. پس از دریافت کامل تمام چانک‌ها، پیام **UPLOAD_FINISH** ثبت می‌شود و Engine با ساخت مانیفست و محاسبه‌ی هش محتوا، یک **CID یکتا** تولید می‌کند که در لاگ با عبارت → **UPLOAD_FINISH CID** نمایش داده می‌شود.این پیام‌ها نشان می‌دهند که آپلود شروع شده و در صورت تکمیل همه‌ی چانک‌ها و موفق بودن ذخیره‌سازی بلاک‌ها، Engine مانیفست را ساخته و CID را تولید کرده است. اگر در زمان ذخیره‌سازی بلاک‌ها خطای فایل‌سیستم رخ دهد، آپلود نباید موفق اعلام شود و Engine باید خطا برگرداند؛ بنابراین "بدون خطا بودن" تنها زمانی درست است که لاگ‌های Engine هیچ پیام خطای ذخیره‌سازی/rename نداشته باشند و پاسخ نهایی CID برگردد.

در لاگ دوم، Gateway پس از دریافت داده‌ی آپلودشده از کلاینت و ارسال آن به Engine، درخواست `POST /upload` را با موفقیت تکمیل کرده و با بازگرداندن وضعیت **OK 200** تأیید می‌کند که فایل بزرگ به‌درستی روی سیستم ذخیره و نهایی‌سازی شده است.

6) تست interrupt

این تست بررسی می‌کند اگر ارتباط HTTP در میانه‌ی آپلود قطع شود، سامانه آپلود را «موفق» اعلام نکند و مانیفست نهایی تولید نشود. در این حالت ممکن است برخی بلاک‌ها قبل از قطع شدن روی دیسک نوشته شده باشند، اما چون همه‌ی چانک‌ها کامل نرسیده‌اند، نباید پاسخ نهایی شامل CID برگردد و نباید فایل manifests/.json ساخته شود. همچنین پس از اجرای مجدد py.main، سرویس باید بدون خطا دوباره بالا بیاید و درخواست‌های جدید را پاسخ دهد. خطای curl با پیام Connection reset by peer نشان می‌دهد اتصال قبل از دریافت پاسخ نهایی از سمت سرور قطع شده است، بنابراین کلاینت نتوانسته CID دریافت کند.

برای این کار باید در ترمینال سوم، دستور زیر را وارد کنیم:

```
dd if=/dev/zero bs=1M count=1000 | \
curl -X POST \
  -H "X-Filename: interrupt.bin" \
  --data-binary @- \
  http://127.0.0.1:9000/upload
```

با انجام این دستور، در ترمینال سوم، شاهد چاپ شدن موارد زیر هستیم:

```
1000+0 records in
1000+0 records out
1048576000 bytes transferred in 0.235458 secs (4453346244 bytes/sec)
curl: (56) Recv failure: Connection reset by peer
```

معنای سه خط اول این است که `dd` واقعاً حدود 1,048,576,000 بایت (1GB) داده تولید و به خروجی استاندارد فرستاده است.
معنای خط اخر هم این است که اتصال TCP/HTTP از سمت سرور به‌صورت ناگهانی بسته شده (سرور یا فرآیند gateway/engine اتصال را reset کرده). در نتیجه `curl` نتوانسته پاسخ نهایی (مثل `cid`) را دریافت کند.

7) Stress Test برای دانلود همزمان

این تست کیس، دو چیز را باهم چک می‌کند:
۱) سرویس زیر بار زیاد کرش نکند، ۲) تعداد threadها کنترل‌شده بماند.
برای این کار، 200 درخواست را **همزمان** (با &) اجرا می‌کند و خروجی هر دانلود را به جای ذخیره‌کردن فایل، داخل `dev/null/` می‌ریزد تا فقط فشار روی سرور/engine ایجاد شود، نه روی دیسک. سپس با `wait` صبر می‌کند همه‌ی درخواست‌ها تمام شوند.
برای انجام این کار، دستور زیر را در ترمینال سوم می‌زنیم:

```
for i in {1..200}; do
  curl -s "http://127.0.0.1:9000/download?cid=34f98f7f8fdee62ac7470dc83e99e23579c42777aa5b80c81f9bac207e077974"
> /dev/null &
done
wait
```

پس از انجام این دستور، نتیجه زیر را در ترمینال سوم می‌بینیم:

```
[2] 13168
[3] 13169
[4] 13170
.
.
.
[19] 13185
[20] 13186
[21] 13187
[7]    done       curl -s  > /dev/null
[7] 13188
[2]    done       curl -s  > /dev/null
[2] 13189
[5]    done       curl -s  > /dev/null
[4]    done       curl -s  > /dev/null
.
.
.
[27]    done       curl -s  > /dev/null
[42]  - done       curl -s  > /dev/null
[4]  + done       curl -s  > /dev/null
[37]  - done       curl -s  > /dev/null
[40]  + done       curl -s  > /dev/null
[20]    done       curl -s  > /dev/null
[35]  + done       curl -s  > /dev/null
[32]  + done       curl -s  > /dev/null
[8]  + done       curl -s  > /dev/null
[5]  + done       curl -s  > /dev/null
```

در اینجا، خطوطی مثل [2]13168 شماره‌ی job و PID هر curl هستند. خطوط done یعنی هر درخواست پس‌زمینه با موفقیت تمام شده و از حلقه خارج شده است. همانطور که مشاهده می‌شود، برنامه کرش نکرده است.
حال برای بررسی اینکه تعداد نخ ها، بی رویه رشد نکرده باشد و کنترل شده مانده باشد، دستور زیر را اجرا میکنیم:

```
ps -M $(pgrep c_engine)
```

اجرای این خط، نتیجه زیر را در همین ترمینال سوم دارد:

```
USER             PID   TT   %CPU STAT PRI     STIME     UTIME COMMAND
melikadehestani  7444 s007    0.0 S    31T   0:00.01   0:00.00 ./c_engine /tmp/cengine.sock
                 7444         0.0 S    31T   0:00.08   0:00.17
                 7444         0.0 S    31T   0:00.08   0:00.17
                 7444         0.0 S    31T   0:00.08   0:00.17
                 7444         0.0 S    31T   0:00.08   0:00.17
```

این نشان دهنده این است که با وجود اجرای 200 دانلود همزمان، تعداد threadهای c_engine محدود و ثابت مانده (چند thread مشخص، نه صدها یا هزاران thread). این یعنی طراحی شما از نظر مدیریت همزمانی (thread pool + محدودسازی اتصال‌ها) درست عمل کرده و دچار thread explosion نشده است.

8) تست (خطا) E_NOT_FOUND

در این تست یک CID با فرمت درست (64 کاراکتر hex) ارسال می‌شود که مانیفست آن روی دیسک وجود ندارد. انتظار داریم Engine خطای E_NOT_FOUND برگرداند و Gateway آن را به پاسخ HTTP مناسب تبدیل کند.

برای این کار، دستور زیر را در ترمینال سوم، وارد می‌کنیم.

```
curl -i “http://127.0.0.1:9000/download?
cid=0000000000000000000000000000000000000000000000000000000000000000”
```

اگر فایل ipfs_store/manifests/.json موجود نباشد، Engine پیام ERROR با code=E_NOT_FOUND ارسال می‌کند و درخواست دانلود ناموفق می‌شود.

```
HTTP/1.0 404 E_NOT_FOUND: manifest not found
Server: OS-Gateway/0.1 Python/3.13.0
Date: Fri, 12 Dec 2025 16:15:58 GMT
Connection: close
Content-Type: text/html;charset=utf-8
Content-Length: 352

<!DOCTYPE HTML>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Error response</title>
    </head>
    <body>
        <h1>Error response</h1>
        <p>Error code: 404</p>
        <p>Message: E_NOT_FOUND: manifest not found.</p>
        <p>Error code explanation: 404 - Nothing matches the given URI.</p>
    </body>
</html>
```

در این صورت به ترتیب در ترمینال main.py و ترمینالی که c_engine.c در حال اجراست، نتایج زیر را می‌بینیم:

```
\\ main.py output

127.0.0.1 - - [12/Dec/2025 19:45:58] code 404, message E_NOT_FOUND: manifest not found
127.0.0.1 - - [12/Dec/2025 19:45:58] "GET /download?
cid=0000000000000000000000000000000000000000000000000000000000000000 HTTP/1.1" 404

\\c_engine.c output

[ENGINE] DOWNLOAD_START: cid="0000000000000000000000000000000000000000000000000000000000000000"
open manifest: No such file or directory
Failed to load manifest for cid 0000000000000000000000000000000000000000000000000000000000000000
```

توضیحات پیاده سازی :

#define _GNU_SOURCE:

این تعریف برای فعال‌سازی قابلیت‌های توسعه‌یافته‌ی کتابخانه‌ی استاندارد C استفاده می‌شود که در برخی سیستم‌ها (به‌ویژه لینوکس) لازم است. در پروژه‌ی ما این خط تضمین می‌کند که توابع و رفتارهای سطح سیستم‌عامل بدون محدودیت در دسترس باشند. سند پروژه استفاده از APIهای سطح سیستم‌عامل را مجاز دانسته و این خط کمک می‌کند پیاده‌سازی Engine در محیط‌های مختلف بدون مشکل کامپایل شود.

#include <semaphore.h>:

این کتابخانه برای استفاده از semaphoreها به‌کار می‌رود که ابزار همزمانی در سیستم‌عامل هستند. در پروژه، Engine باید بتواند چندین اتصال همزمان را مدیریت کند، اما بدون ایجاد بی‌نهایت thread. ما با استفاده از semaphore این نیاز پروژه را به‌صورت کنترل‌شده پیاده‌سازی کرده‌ایم.

#define MAX_CONNECTIONS 64, static sem_t conn_sem:

این دو خط حداکثر تعداد اتصال‌های همزمان به Engine را مشخص می‌کنند. طبق پروژه، Engine باید همزمانی را پشتیبانی کند ولی پایدار بماند. ما با تعیین سقف اتصال و استفاده از conn_sem تضمین کرده‌ایم که هر اتصال جدید فقط در صورت وجود ظرفیت پردازش پذیرفته شود و از overload شدن سیستم جلوگیری شود.

#include <sys/socket.h>, #include <sys/un.h>:

این کتابخانه‌ها برای پیاده‌سازی Unix Domain Socket (AF_UNIX) استفاده می‌شوند. سند پروژه صراحتاً ارتباط بین Gateway و Engine را از طریق Unix socket محلی الزام کرده است. ما دقیقاً مطابق سند از این کتابخانه‌ها برای ایجاد، bind و listen روی سوکت استفاده کرده‌ایم.

#include <sys/types.h>, #include <sys/stat.h>, #include <unistd.h>:

این کتابخانه‌ها برای عملیات پایه‌ی سیستم‌عامل مانند کار با فایل‌ها، دایرکتوری‌ها و descriptorها استفاده می‌شوند. پروژه نیاز دارد که Engine بتواند فایل‌ها و دایرکتوری‌های ذخیره‌سازی را مدیریت کند. این includeها ابزار لازم برای اجرای این بخش از پروژه را فراهم می‌کنند.

#include <arpa/inet.h>:

این کتابخانه برای تبدیل ترتیب بایت‌ها بین host و network استفاده می‌شود. در سند پروژه، طول پیام در فریم باید به‌صورت big-endian ارسال شود. ما با استفاده از htonl و ntohl این الزام پروتکل را به‌درستی پیاده‌سازی کرده‌ایم.

#include <errno.h>:

این کتابخانه برای تشخیص دقیق نوع خطاهای سیستمی استفاده می‌شود. در پروژه لازم است خطاهای مختلف (مانند قطع ارتباط یا خطای فایل) به‌درستی مدیریت شوند. استفاده از errno باعث می‌شود رفتار Engine در شرایط خطا قابل پیش‌بینی و قابل اشکال‌زدایی باشد.

#include <pthread.h>:

این کتابخانه برای پیاده‌سازی threadها، mutex و condition variable استفاده شده است. سند پروژه نیاز به پردازش همزمان آپلود و دانلود را مشخص کرده است. ما با استفاده از pthreadها این همزمانی را به‌صورت واقعی و ایمن پیاده‌سازی کرده‌ایم.

#include <inttypes.h> و #include <stdint.h>

این کتابخانه‌ها برای استفاده از نوع‌های عددی دقیق مانند uint32_t و uint64_t به‌کار رفته‌اند. در پروژه، اندازه‌ی فریم‌ها، chunk index و total size باید دقیق و مستقل از معماری سیستم باشند. این includeها تضمین می‌کنند که مقادیر در همه‌ی سیستم‌ها یکسان تفسیر شوند.

#include <stdlib.h> و #include <stdio.h>

برای ورودی/خروجی استاندارد، تخصیص حافظه و مدیریت منابع استفاده می‌شوند. در پروژه برای لاگ‌گیری، ساخت JSON مانیفست و تخصیص حافظه‌ی چانک‌ها ضروری هستند.

#include <limits.h> و #include <string.h>

این کتابخانه‌ها برای کار با رشته‌ها و محدودیت‌های سیستم (مثل PATH_MAX) استفاده شده‌اند. پروژه نیاز به ساخت مسیر فایل‌ها و پردازش رشته‌هایی مانند filename و CID دارد که این includeها را ضروری می‌کند.

#include <fcntl.h>

برای کار با پرچم‌های فایل مانند O_CREAT, O_TRUNC و O_RDWR استفاده شده است. در پروژه، ذخیره‌ی بلاک‌ها و مانیفست‌ها نیازمند کنترل دقیق نحوه‌ی باز شدن فایل‌هاست که با این کتابخانه انجام شده است.

#include <sys/file.h>

این کتابخانه برای استفاده از flock و قفل‌گذاری روی فایل‌ها به‌کار رفته است. در پروژه ممکن است چند thread همزمان به refcount یک بلاک دسترسی داشته باشند. ما با استفاده از این ابزار، الزام سند پروژه برای جلوگیری از race condition را رعایت کرده‌ایم.

#include "blake3/blake3.h"

این کتابخانه پیاده‌سازی الگوریتم هش BLAKE3 را فراهم می‌کند. سند پروژه صراحتاً استفاده از BLAKE3 را الزام کرده است. ما از این کتابخانه هم برای هش چانک‌ها و هم برای تولید CID مانیفست استفاده کرده‌ایم.

تعریف Opcodeها

این مقادیر عددی نوع پیام‌های ردوبدل‌شده بین Gateway و Engine را مشخص می‌کنند. سند پروژه پروتکل پیام و opcodeهای مربوط به آپلود، دانلود و خطا را تعریف کرده است. ما این opcodeها را به‌صورت ثابت تعریف کرده‌ایم تا هر پیام دقیقاً مطابق قرارداد پروتکل تفسیر شود.

#define STORAGE_ROOT, BLOCKS_DIR, MANIFESTS_DIR

این تعریف‌ها مسیر ساختار ذخیره‌سازی Engine را مشخص می‌کنند. طبق سند پروژه، بلاک‌ها و مانیفست‌ها باید در مسیرهای جداگانه ذخیره شوند. ما این ساختار را دقیقاً مطابق توضیح پروژه پیاده‌سازی کرده‌ایم.

#define DEFAULT_CHUNK_SIZE (256 * 1024)

این مقدار اندازه‌ی پیش‌فرض هر چانک را تعیین می‌کند. سند پروژه به‌طور صریح اندازه‌ی چانک ۲۵۶ کیلوبایت را الزام کرده است. این مقدار باید با Gateway یکسان باشد و در مانیفست نیز ثبت می‌شود.

#define NUM_WORKERS 4

این مقدار تعداد threadهای worker در thread pool را مشخص می‌کند. پروژه نیاز به پردازش موازی چانک‌ها دارد. ما با استفاده از thread pool ثابت، پردازش موازی را بدون ایجاد بی‌رویه‌ی thread پیاده‌سازی کرده‌ایم.

#define MAX_FRAME (4 * 1024 * 1024)

این مقدار حداکثر اندازه‌ی payload پیام‌ها را مشخص می‌کند. سند پروژه اعتبارسنجی پیام‌ها را الزامی کرده است. ما با این محدودیت از تخصیص حافظه‌ی بیش‌ازحد و حملات احتمالی جلوگیری کرده‌ایم.

#define MAX_NAME_LEN 4096

حداکثر طول مجاز برای filename یا CID را تعیین می‌کند. این بررسی بخشی از اعتبارسنجی ورودی‌ها طبق سند پروژه است تا از پیام‌های مخرب یا اشتباه جلوگیری شود.

static const char* g_sock_path

این متغیر مسیر Unix socket را که از آرگومان خط فرمان دریافت می‌شود نگه می‌دارد. سند پروژه اجرای Engine با مسیر مشخص سوکت را الزام کرده و این متغیر پیاده‌سازی مستقیم همان نیاز است.

تابع die

این تابع برای مدیریت خطاهای بحرانی سیستم‌عامل استفاده می‌شود. ورودی آن یک رشته‌ی توضیح خطاست و خروجی ندارد، زیرا برنامه را با (exit1) متوقف می‌کند. در پروژه، طبق سند، اگر خطاهایی مانند شکست در bind، socket یا listen رخ دهد، ادامه‌ی اجرا بی‌معنی است و Engine باید متوقف شود. ما این الزام را با استفاده از perror برای چاپ خطای سیستمی و خروج فوری پیاده‌سازی کرده‌ایم تا خطاها شفاف و قابل تشخیص باشند.

تابع read_n

این تابع مسئول خواندن دقیقاً n بایت از یک file descriptor (معمولاً سوکت) است. ورودی‌های آن شامل descriptor، بافر مقصد و تعداد بایت مورد انتظار است و خروجی آن تعداد بایت خوانده‌شده یا خطا است. در سند پروژه تأکید شده که پیام‌ها به‌صورت فریم‌شده (opcode + length + payload) هستند و نباید فرض کنیم که read همه‌ی داده را یک‌جا برمی‌گرداند. این تابع با حلقه و مدیریت EINTR تضمین می‌کند که payload کامل پیام دریافت شود و پروتکل به‌درستی پیاده‌سازی شود.

تابع write_all

این تابع برای نوشتن کامل داده روی سوکت استفاده می‌شود. ورودی آن descriptor، بافر داده و طول داده است و خروجی آن موفقیت یا خطاست. مشابه read_n، این تابع تضمین می‌کند که کل پیام (header یا payload) ارسال شود و نوشتن ناقص رخ ندهد. این دقیقاً مطابق نیاز سند پروژه برای ارسال پیام‌های کامل در IPC است و از شکسته‌شدن فریم‌های پروتکل جلوگیری می‌کند.

تابع send_frame

این تابع هسته‌ی پیاده‌سازی پروتکل پیام Engine است. ورودی‌های آن شامل opcode، file descriptor، پیام payload و طول payload است. این تابع طبق سند پروژه یک فریم استاندارد می‌سازد که شامل ۱ بایت ۴ ،opcode بایت طول payload به صورت big-endian و سپس payload است. ما با استفاده از htonl ترتیب بایت‌ها را استاندارد کرده‌ایم تا Gateway و Engine دقیقاً مطابق تعریف پروتکل با هم ارتباط برقرار کنند.

تابع send_error

این تابع برای ارسال پیام‌های خطا به Gateway استفاده می‌شود. ورودی آن descriptor، کد خطا (مثل E_PROTO) و پیام توضیحی است. خروجی ندارد و پیام را در قالب JSON و با opcode برابر 0xFF ارسال می‌کند. سند پروژه صراحتاً وجود پیام خطا با opcode مشخص و payload متنی را الزام کرده است و ما این را با قالب JSON ساده و محدودسازی طول پیام پیاده‌سازی کرده‌ایم تا هم استاندارد باشد و هم ایمن.

تابع ensure_dir

این تابع بررسی می‌کند که یک دایرکتوری وجود دارد یا نه و در صورت نبود، آن را ایجاد می‌کند. ورودی آن مسیر دایرکتوری و خروجی آن موفقیت یا خطاست. در پروژه، قبل از ذخیره‌ی بلاک‌ها و مانیفست‌ها باید ساختار ipfs_store, blocks, و manifests وجود داشته باشد. این تابع پیاده‌سازی مستقیم همین الزام سند پروژه است و اجازه می‌دهد Engine بدون نیاز به آماده‌سازی دستی اجرا شود.

تابع ensure_parents_for_path

این تابع تمام دایرکتوری‌های والد یک مسیر فایل را به‌ترتیب ایجاد می‌کند. ورودی آن مسیر کامل فایل است و خروجی آن وضعیت موفقیت است. چون در پروژه بلاک‌ها در مسیرهایی مانند blocks/aa/bb/hash.bin ذخیره می‌شوند، لازم است تمام دایرکتوری‌های میانی وجود داشته باشند. سند پروژه به ساختار سلسله‌مراتبی ذخیره‌سازی اشاره دارد و ما با این تابع آن را به‌صورت پویا و امن پیاده‌سازی کرده‌ایم.

تابع write_file_atomic

این تابع نوشتن اتمیک فایل را پیاده‌سازی می‌کند. ورودی‌های آن مسیر فایل، داده و طول داده هستند. طبق سند پروژه، مانیفست و داده‌ها نباید به‌صورت ناقص روی دیسک دیده شوند. ما ابتدا داده را در فایل موقت (.tmp) می‌نویسیم، با fsync آن را روی دیسک تثبیت می‌کنیم و سپس با rename جایگزین می‌کنیم. این روش تضمین می‌کند که یا فایل کاملاً نوشته می‌شود یا اصلاً نوشته نمی‌شود.

تابع blake3_hex

این تابع داده‌ی ورودی را با الگوریتم BLAKE3 هش می‌کند و خروجی را به‌صورت رشته‌ی hex برمی‌گرداند. ورودی‌های آن بافر داده و طول آن است و خروجی در آرایه‌ی out_hex قرار می‌گیرد. سند پروژه استفاده از BLAKE3 را الزام کرده و ما با استفاده از API رسمی BLAKE3 این هش را برای چانک‌ها و همچنین تولید CID مانیفست استفاده کرده‌ایم.

تابع make_block_path

این تابع مسیر ذخیره‌سازی یک بلاک را بر اساس hash آن می‌سازد. ورودی آن hash به صورت hex و خروجی مسیر کامل فایل است. طبق سند پروژه، ذخیره‌سازی باید content-addressed و به‌صورت دایرکتوری‌بندی‌شده انجام شود. ما با استفاده از دو بایت اول hash، مسیر blocks/aa/bb/hash.bin را ایجاد کرده‌ایم که هم مقیاس‌پذیر است و هم مطابق ساختار خواسته‌شده.

تابع make_refcount_path

این تابع مسیر فایل refcount مربوط به هر بلاک را می‌سازد. ورودی آن hash بلاک و خروجی مسیر فایل .ref است. در پروژه برای جلوگیری از حذف زودهنگام بلاک‌های مشترک، نیاز به نگه‌داشتن تعداد ارجاعات وجود دارد. این تابع پایه‌ی پیاده‌سازی deduplication و مدیریت بلاک‌های مشترک است.

تابع inc_refcount

این تابع مقدار refcount یک بلاک را به‌صورت ایمن افزایش می‌دهد. ورودی آن hash بلاک است و خروجی ندارد. طبق سند پروژه و الزامات همزمانی، ممکن است چند thread همزمان به یک بلاک دسترسی داشته باشند. ما با استفاده از flock قفل فایل را گرفته‌ایم تا از race condition جلوگیری کنیم و مقدار refcount همیشه صحیح باقی بماند.

تابع make_manifest_path

این تابع مسیر فایل مانیفست مربوط به یک CID را تولید می‌کند. ورودی آن CID و خروجی مسیر <cid.json> در پوشه‌ی manifests است. این دقیقاً مطابق سند پروژه است که مشخص کرده مانیفست‌ها باید با نام CID ذخیره شوند و Engine بتواند در زمان دانلود آن‌ها را پیدا کند.

ChunkMeta struct

ChunkMeta یک ساختار متادیتا برای هر چانک است که شامل index (شماره چانک)، size (اندازه واقعی چانک) و hash (هش ۶۴-کاراکتری) می‌باشد. این دقیقاً همان چیزی است که سند پروژه برای مانیفست الزام کرده: آرایه‌ای از چانک‌ها با فیلدهای index, size, hash تا بازسازی و راستی‌آزمایی ساده شود. ما این ساختار را به عنوان "فرمت داخلی" نگه می‌داریم و بعداً هنگام ساخت مانیفست JSON، همین فیلدها را خروجی می‌دهیم.

UploadState struct

UploadState وضعیت آپلود جاری را نگه می‌دارد: اینکه آپلود فعال است یا نه (active)، نام فایل (filename)، اندازه کل (total_size)، اندیس چانک بعدی (next_index)، تعداد کل چانک‌ها (total_chunks) و آرایه‌ی متادیتای چانک‌ها (chunks/chunks_cap). همچنین برای همزمانی، pending_tasks تعداد تسک‌های درحال پردازش را نگه می‌دارد و با pthread_mutex_t mu و pthread_cond_t cv هماهنگی انجام می‌شود؛ had_error هم برای این است که اگر ذخیره‌سازی یک بلاک fail شد، پایان آپلود "موفق" اعلام نشود. سند پروژه گفته Engine باید هنگام آپلود چانک‌ها را موازی پردازش کند و در پایان مانیفست بسازد و CID را برگرداند؛ این State دقیقاً همان داده‌های لازم برای ساخت مانیفست و کنترل پایان آپلود را فراهم می‌کند.

تابع upload_state_construct

این تابع ساخت اولیه‌ی UploadState را انجام می‌دهد. ورودی یک اشاره‌گر به state است و خروجی ندارد؛ ابتدا با memset همه چیز صفر می‌شود و سپس mutex و condvar مقداردهی اولیه می‌شوند. نکته‌ی کلیدی این است که بدون این init، همزمانی درست کار نمی‌کند. این کار در راستای بخش "مکانیسم‌های همگام‌سازی" سند است که می‌گوید برای مدیریت رقابت و بیدارباش کارآمد از mutex و condition variable استفاده کنید.

تابع upload_state_reset

این تابع وضعیت آپلود را برای شروع یک آپلود جدید یا پاک‌سازی پس از پایان آماده می‌کند. ورودی state است، خروجی ندارد؛ درون mutex قفل می‌گیرد، حافظه‌ی filename و chunks را آزاد می‌کند و همه‌ی شمارنده‌ها (مثل total_size, next_index, pending_tasks, had_error) را ریست می‌کند. از نظر پروژه، چون Gateway برای هر درخواست یک

فرآیند جداست ولی Engine می‌تواند چند پیام یک اتصال را مدیریت کند، لازم است بعد از UPLOAD_FINISH state پاک شود تا آپلود بعدی در همان اتصال، داده‌ی قبلی را قاطی نکند؛ این دقیقاً برای "پایداری و صحت" جریان آپلود ضروری است.

تابع upload_state_destroy

این تابع تخریب نهایی state را انجام می‌دهد: اول upload_state_reset را صدا می‌زند تا حافظه آزاد شود و سپس mutex و condvar را destroy می‌کند. ورودی state است و خروجی ندارد. این کار برای جلوگیری از memory leak و باقی ماندن منابع همزمانی لازم است، مخصوصاً وقتی اتصال بسته می‌شود یا handler تمام می‌شود.

DownloadSlot struct

DownloadSlot یک بافر/اسلات برای نتیجه‌ی آماده‌شده‌ی هر چانک در دانلود است: data (بافر چانک)، size (اندازه)، ready (آماده‌شدن) و error (کد خطا مثل نبودن چانک یا mismatch). ایده این است که workerها چانک‌ها را ممکن است خارج از ترتیب آماده کنند، اما handler دانلود باید بتواند به ترتیب index منتظر بماند و خروجی را درست بچیند. این دقیقاً مطابق سند است که می‌گوید ادغام خروجی دانلود باید ترتیبی باشد حتی اگر پردازش موازی و خارج از ترتیب تکمیل شود.

DownloadState struct

DownloadState وضعیت دانلود یک CID را نگه می‌دارد: تعداد چانک‌ها (n_chunks)، آرایه‌ی slotها (slots) و ابزار همزمانی (mu, cv) برای اینکه thread اصلی دانلود بتواند منتظر آماده شدن هر index بماند. سند پروژه تأکید دارد که هر چانک پس از verify موفق ارسال شود و ترتیب نهایی فایل درست باشد؛ این State همان زیرساخت لازم برای "پردازش موازی + ادغام ترتیبی" را فراهم می‌کند.

تابع download_state_construct

این تابع سازنده‌ی DownloadState است. ورودی‌ها اشاره‌گر به state و تعداد چانک‌هاست؛ خروجی ندارد. ابتدا state صفر می‌شود، سپس slots با calloc به اندازه‌ی n_chunks ساخته می‌شود و mutex و condvar مقداردهی اولیه می‌شوند. نکته کلیدی این است که چون handler باید از روی index منتظر بماند، باید از ابتدا به تعداد چانک‌های مانیفست اسلات داشته باشیم تا هر worker نتیجه را در جای درست بنویسد؛ این دقیقاً مطابق مکانیسم‌های همگام‌سازی و الگوی ادغام ترتیبی سند است.

تابع download_state_destroy

این تابع آزادسازی منابع DownloadState را انجام می‌دهد. ورودی state است و خروجی ندارد؛ ابتدا mutex قفل می‌شود، سپس برای هر slot اگر داده‌ای تخصیص یافته آزاد می‌شود، آرایه‌ی slots آزاد می‌شود و در پایان mutex و condvar destroy می‌شوند. این کار برای جلوگیری از memory leak در دانلودهای همزمان لازم است (چون طبق سند چند دانلود همزمان از یک CID ممکن است انجام شود) و باعث می‌شود هر دانلود مستقل، منابع خودش را تمیز جمع کند.

تعریف TaskType

این enum نوع تسک را مشخص می‌کند تا worker بفهمد این تسک مربوط به آپلود است یا دانلود. دو مقدار TASK_UPLOAD و TASK_DOWNLOAD دارید که مسیر اجرای worker را تعیین می‌کند. این دقیقاً همان "برچسب‌گذاری کارها"ست تا یک صف واحد بتواند چند نوع کار را حمل کند.

تعریف Task

ساختار Task یک «واحد کار» در صف است. یک فیلد type دارد که می‌گوید آپلود است یا دانلود، یک next دارد تا داخل صف لینک‌شده قرار بگیرد، و یک union دارد که بسته به نوع تسک، داده‌های مورد نیاز را نگه می‌دارد. برای آپلود، آدرس UploadState، شماره‌ی چانک، بافر داده و اندازه نگه داشته می‌شود؛ برای دانلود، آدرس DownloadState، شماره‌ی چانک و هش چانک نگه داشته می‌شود. این پیاده‌سازی دقیقاً نیاز پروژه را برای «صف کردن کارها و اجرای موازی در workerها» برآورده می‌کند و باعث می‌شود state اشتراکی (UploadState/DownloadState) به‌صورت کنترل‌شده بین نخ‌ها استفاده شود.

متغیرهای صف تسک‌ها (q_head, q_tail, q_mu, q_cv)

q_head و q_tail سر و ته صف لینک‌شده‌ی تسک‌ها هستند. q_mu یک mutex سراسری است که تضمین می‌کند اضافه/کم کردن از صف همزمان خراب نشود (race نخورید). q_cv شرط (condition variable) است تا workerها وقتی صف خالی است sleep کنند و وقتی producer یک تسک اضافه کرد بیدار شوند. این دقیقاً همان چیزی است که پروژه از «همگام‌سازی صحیح با mutex + condvar» انتظار دارد.

تابعenqueue_task

این تابع نقش "producer" را دارد: تسک ساخته‌شده را وارد صف می‌کند. ابتدا با mutex صف را قفل می‌کند تا دو نخ همزمان صف را خراب نکنند، سپس تسک را به انتهای صف وصل می‌کند (با q_tail) و در پایان با pthread_cond_signal یک worker منتظر را بیدار می‌کند و mutex را آزاد می‌کند. این تابع بخش مهمی از خواسته‌ی پروژه برای همگام‌سازی است چون تضمین می‌کند دسترسی به صف thread-safe باشد و سیستم بی‌دلیل busy-wait نکند.

تابع process_upload_task

این تابع "کار واقعی آپلود" را انجام می‌دهد: داده‌ی چانک را هش می‌کند (BLAKE3)، مسیر بلاک را از روی هش می‌سازد، و اگر بلاک از قبل وجود نداشت آن را با نوشتن اتمیک ذخیره می‌کند. اگر نوشتن بلاک شکست بخورد، داخل UploadState پرچم had_error ست می‌شود تا در UPLOAD_FINISH آپلود به‌عنوان شکست‌خورده اعلام شود؛ اگر موفق باشد inc_refcount صدا زده می‌شود تا رفرنس بلاک افزایش یابد. سپس متادیتای چانک (index/size/hash) داخل آرایه‌ی chunks در UploadState ثبت می‌شود (و اگر ظرفیت کافی نباشد realloc انجام می‌دهد)، بعد شمارنده‌ی pending_tasks کم می‌شود و با pthread_cond_broadcast نخ منتظر در UPLOAD_FINISH بیدار می‌شود. در نهایت بافر داده آزاد می‌شود چون مالکیت آن از لحظه‌ی صف شدن تسک، به تسک منتقل شده بود. این تابع دقیقاً بخش پروژه درباره‌ی «هش و ذخیره چانک‌ها + مدیریت همزمانی + آماده‌سازی برای ساخت مانیفست» را پوشش می‌دهد.

تابع process_download_task

این تابع "کار واقعی دانلود" را برای یک چانک انجام می‌دهد: از روی هش چانک مسیر بلاک را می‌سازد، فایل بلاک را باز می‌کند و کامل داخل حافظه می‌خواند، سپس دوباره هش می‌گیرد و با هش مورد انتظار مقایسه می‌کند تا صحت داده تأیید شود. نتیجه در DownloadState داخل slots[index] قرار می‌گیرد: اگر فایل نبود یا خطای I/O رخ دهد error=1، اگر mismatch باشد error=2، و اگر درست بود بافر و اندازه در slot ذخیره می‌شود و ready=1 می‌شود. بعد با pthread_cond_broadcast نخ merger که منتظر آماده شدن آن slot است بیدار می‌شود. نکته‌ی کلیدی این است که این تابع فقط "آماده‌سازی" را انجام می‌دهد و ترتیب ارسال را تعیین نمی‌کند؛ ترتیب ارسال در merger (که index به index منتظر می‌ماند) تضمین می‌شود. این دقیقاً با خواسته‌ی پروژه درباره‌ی «راستی‌آزمایی هنگام دانلود + امکان پردازش موازی اما خروجی ترتیبی» منطبق است.

تابع worker_main

این تابع حلقه‌ی بی‌نهایت workerهاست. هر worker با mutex صف را قفل می‌کند و تا وقتی صف خالی است روی pthread_cond_wait می‌خوابد؛ وقتی تسکی اضافه شد، یک تسک را از سر صف برمی‌دارد، سر/ته صف را آپدیت می‌کند و mutex را آزاد می‌کند. سپس بر اساس نوع تسک، یا process_upload_task را صدا می‌زند یا process_download_task را، و در پایان خودِ Task را آزاد می‌کند. این همان موتور اصلی thread pool است که باعث می‌شود چند چانک همزمان پردازش شوند و نخ اتصال فقط نقش دریافت پیام‌ها و صف‌کردن کارها را داشته باشد، نه اجرای مستقیم کار سنگین.

تابع build_manifest_json

این تابع پس از پایان موفق آپلود فراخوانی می‌شود و وظیفه‌ی آن ساخت فایل مانیفست JSON از وضعیت UploadState است. ابتدا با open_memstream یک بافر حافظه‌ای می‌سازد تا JSON به‌صورت متنی در آن نوشته شود. سپس با قفل گرفتن روی UploadState، اطلاعاتی مانند نسخه‌ی مانیفست، الگوریتم هش (blake3)، اندازه‌ی چانک ثابت، اندازه‌ی کل فایل، نام فایل و لیست چانک‌ها (index، size و hash هر چانک) را در قالب JSON می‌نویسد. بعد از بسته‌شدن stream، روی کل متن مانیفست هش BLAKE3 گرفته می‌شود و خروجی آن به‌عنوان CID استفاده می‌گردد. این تابع دقیقاً بخش پروژه درباره‌ی «ساخت مانیفست و محاسبه‌ی CID» را پوشش می‌دهد و تضمین می‌کند هر تغییر در چانک‌ها باعث تغییر CID شود.

تابع manifest_chunks_free

این تابع مسئول آزادسازی حافظه‌ی ساختار ManifestChunks است که در زمان دانلود استفاده می‌شود. پس از اتمام دانلود یا در صورت بروز خطا، آرایه‌ی چانک‌ها آزاد می‌شود و شمارنده‌ها ریست می‌گردند. این تابع از نشت حافظه جلوگیری می‌کند و تکمیل‌کننده‌ی چرخه‌ی عمر داده‌های مانیفست است، هرچند نقش مفهومی مستقیمی در متن پروژه ندارد ولی برای پیاده‌سازی صحیح ضروری است.

تابع json_expect_int

این تابع یک helper ساده برای استخراج مقدار عددی از JSON مانیفست است. با جستجوی کلید مشخص‌شده (مثل version یا chunk_size) در متن JSON، مقدار عددی بعد از آن را می‌خواند و در خروجی می‌ریزد. در فرآیند دانلود، از این تابع برای اعتبارسنجی مانیفست استفاده می‌شود تا اطمینان حاصل شود نسخه و اندازه‌ی چانک با انتظارات Engine مطابقت دارد. این تابع بخشی از پیاده‌سازی بررسی صحت مانیفست طبق پروژه است.

تابع json_expect_string

این تابع مشابه json_expect_int است اما برای استخراج رشته‌ها از JSON (مثل hash_algo). در زمان خواندن مانیفست، از آن استفاده می‌شود تا بررسی شود الگوریتم هش استفاده‌شده همان blake3 است. این کار مانع استفاده از مانیفست ناسازگار یا خراب می‌شود و نقش آن در پروژه، اعتبارسنجی فراداده‌ی مانیفست قبل از دانلود است.

تابع load_manifest_chunks

این تابع هسته‌ی بخش دانلود است و وظیفه‌ی آن بارگذاری و اعتبارسنجی مانیفست بر اساس CID می‌باشد. ابتدا مسیر فایل مانیفست را از روی CID می‌سازد و فایل JSON را کامل در حافظه می‌خواند. سپس با استفاده از helperها، فیلدهای حیاتی مثل نسخه، اندازه‌ی چانک و الگوریتم هش را بررسی می‌کند و اگر ناسازگار باشند، خطا می‌دهد. بعد وارد آرایه‌ی chunks می‌شود و برای هر چانک، هش آن را استخراج کرده و در ساختار ManifestChunks ذخیره می‌کند. این تابع طبق پروژه تضمین می‌کند که

Engine بداند چند چانک باید دانلود شود و هش هر کدام چیست تا در مرحله‌ی بعد، صحت داده‌ها verify شود. این بخش مستقیماً الزام پروژه درباره‌ی «دانلود مبتنی بر مانیفست و بررسی صحت چانک‌ها» را پیاده‌سازی می‌کند.

تابع queue_upload_chunk

این تابع وظیفه‌ی صف‌کردن یک چانک آپلود را بر عهده دارد. پس از دریافت payload مربوط به یک UPLOAD_CHUNK، یک Task جدید از نوع TASK_UPLOAD ساخته می‌شود و مالکیت بافر داده به آن منتقل می‌گردد. سپس با قفل گرفتن روی UploadState، شماره‌ی چانک به‌صورت ترتیبی تولید می‌شود (با افزایش next_index)، تعداد کل چانک‌ها و total_size به‌روزرسانی می‌شود و pending_tasks افزایش می‌یابد تا بعداً در UPLOAD_FINISH بتوان منتظر اتمام همه‌ی کارها ماند. در نهایت تسک در صف مشترک workerها قرار می‌گیرد. این تابع دقیقاً خواسته‌ی پروژه درباره‌ی پردازش همزمان چانک‌ها و مدیریت thread-safe وضعیت آپلود را پیاده‌سازی می‌کند.

تابع handle_connection

این تابع برای هر اتصال Gateway به Engine اجرا می‌شود و مسئول پردازش کامل پیام‌های پروتکل روی آن اتصال است. ابتدا یک UploadState ساخته می‌شود و سپس در یک حلقه، هدر فریم‌ها (opcode و طول payload) خوانده می‌شود، طول payload اعتبارسنجی می‌گردد و payload دریافت می‌شود. در صورت دریافت OP_UPLOAD_START، وضعیت قبلی ریست می‌شود، آپلود فعال می‌گردد و نام فایل پس از بررسی طول ذخیره می‌شود. در OP_UPLOAD_CHUNK، اگر آپلود فعال نباشد خطای پروتکل برگردانده می‌شود و در غیر این صورت چانک با استفاده از queue_upload_chunk وارد صف پردازش می‌شود. در OP_UPLOAD_FINISH، تابع تا پایان تمام تسک‌های آپلود منتظر می‌ماند، خطاهای احتمالی ذخیره‌سازی را بررسی می‌کند، از کامل بودن همه‌ی چانک‌ها مطمئن می‌شود، سپس مانیفست را می‌سازد، CID را محاسبه می‌کند، مانیفست را به‌صورت اتمیک ذخیره می‌کند و در نهایت OP_UPLOAD_DONE را همراه CID ارسال می‌کند. این قسمت الزام پروژه برای «اعلام موفقیت فقط پس از اتمام واقعی آپلود» و «نوشتن اتمیک مانیفست» را پوشش می‌دهد.

در مسیر دانلود، با دریافت START_DOWNLOAD_OP ابتدا CID از نظر فرمت بررسی می‌شود و در صورت نامعتبر بودن، خطای E_BAD_CID از طرف Engine ارسال می‌گردد. سپس مانیفست از مسیر ipfs_store/manifests/.json بارگذاری می‌شود و اگر فایل مانیفست وجود نداشته باشد، خطای E_NOT_FOUND برگردانده می‌شود. همچنین اگر هنگام بازیابی هر چانک، فایل بلاک وجود نداشته باشد E_NOT_FOUND و اگر هش محتوای چانک با هش ثبت‌شده در مانیفست مغایرت داشته باشد E_HASH_MISMATCH ارسال می‌شود. این خطاها در قالب پیام ERROR (opcode=0xFF) با payload شامل code و message به Gateway منتقل می‌گردند.

بعد برای هر چانک، یک تسک دانلود ساخته و در صف workerها قرار داده می‌شود. پس از آن، یک مرحله‌ی sequential merger اجرا می‌شود که index به index منتظر آماده شدن هر چانک می‌ماند و در صورت صحت داده، آن را به ترتیب با OP_DOWNLOAD_CHUNK ارسال می‌کند. اگر خطایی مثل نبودن بلاک یا عدم تطابق هش رخ دهد، دانلود متوقف شده و خطای مناسب ارسال می‌شود. در پایان موفقیت‌آمیز، OP_DOWNLOAD_DONE فرستاده می‌شود. این بخش دقیقاً نیاز پروژه برای «دانلود موازی با خروجی ترتیبی و بررسی صحت چانک‌ها» را پیاده‌سازی می‌کند.

در انتهای تابع، منابع آزاد می‌شوند، وضعیت آپلود تخریب می‌گردد، semaphore اتصال آزاد می‌شود و سوکت بسته می‌شود تا کنترل همزمانی اتصال‌ها رعایت شود.

تابع main

این تابع نقطه‌ی شروع Engine است و کل زیرساخت اجرایی پروژه را راه‌اندازی می‌کند. ابتدا مسیر سوکت یونیکس از آرگومان دریافت می‌شود و دایرکتوری‌های ذخیره‌سازی (blocks و manifests) ساخته می‌شوند. سپس thread pool سراسری workerها ایجاد و detach می‌شود تا پردازش تسک‌ها به‌صورت همزمان انجام گیرد. semaphore برای محدود کردن تعداد

اتصال‌های همزمان مقداردهی می‌شود. بعد سوکت AF_UNIX ساخته، bind و listen می‌شود و Engine وارد حلقه‌ی accept می‌گردد. برای هر اتصال جدید، یک thread جداگانه ایجاد می‌شود که handle_connection را اجرا می‌کند. این طراحی دقیقاً معماری خواسته‌شده‌ی پروژه را پیاده‌سازی می‌کند: یک Engine سروری مستقل که از طریق IPC با Gateway ارتباط دارد، چندین اتصال را همزمان مدیریت می‌کند و کارهای سنگین را به worker threadها می‌سپارد.

نتیجه گیری

در پایان، این پروژه نشان داد که چگونه می‌توان با استفاده از مفاهیم پایه‌ای سیستم‌عامل، یک سامانه‌ی نسبتاً پیچیده و نزدیک به سیستم‌های واقعی را به‌صورت عملی پیاده‌سازی کرد. در طول این کار، جداسازی فرایندی بین Gateway و Engine، استفاده از ارتباط بین‌فرایندی با Unix Domain Socket، مدیریت همزمانی با نخ‌ها و مکانیزم‌های همگام‌سازی، و کنترل دقیق خطاها نقش کلیدی در پایداری و صحت عملکرد سامانه داشتند. اگرچه برخی بخش‌ها مانند Merkle-DAG و CID به‌صورت ساده‌سازی‌شده پیاده‌سازی شدند، اما اهداف اصلی پروژه که درک معماری محتوامحور، پردازش همزمان، و تضمین صحت داده‌ها بود به‌خوبی محقق شد. این تجربه دید عملی روشنی از چالش‌های طراحی سیستم‌های سطح پایین، تصمیم‌گیری‌های مهندسی بین سادگی و استانداردبودن، و اهمیت مستندسازی شفاف محدودیت‌ها و انتخاب‌های طراحی فراهم کرد و پایه‌ی مناسبی برای درک و توسعه‌ی سیستم‌های پیشرفته‌تر در آینده ایجاد نمود.

لینک های مفید

- برای دریافت فایل های پروژه به گیت هاب مراجعه نمایید.
- برای دریافت ادیتور کدها به سایت carbon مراجعه نمایید.
- جهت اشنایی با منطق IPFS به ویدیو یوتیوب مراجعه نمایید.