Part 1

Project Understanding

The objective of this project is to implement a web proxy server, using libcurl library to fetch requested files from HTTP servers.

In order to convert the getfile server into a proxy server:

- Instead of retrieving file from disc, I used libcurl's "easy" interface to perform the file downloading from HTTP servers.
- The URL is specified by concatenating the server name and file path.
- I set a memory space and callbacks for libcurl to download the file chunk by chunk.
- CURLOPT_FAILONERROR option was set, which tells the library to fail the request if the HTTP code returned is equal to or larger than 400. On a 404 from the webserver, proxy returns a GF_FILE_NOT_FOUND header to client.
- I used the curl_easy_getinfo function to request the CURLINFO_SIZE_DOWNLOAD_T information from the curl session, which return the number of bytes downloaded.

Flow of Control

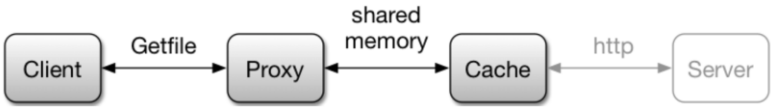
1. Initialize libcurl
2. Loop foever:
   - Boss thread:
     Receive requests from clients
   - Each worker thread:
     Start a libcurl session, specify libcurl options and callbacks
     Perform the transfer as described in the options
     Get the length of downloaded data, send header to client
     Send the downloaded data to client
     End the libcurl session
3. Cleanup libcurl

Testing

- Run proxy process with different thread numbers
- Run client process with different thread numbers, request numbers
- Request for nonexistent files

Part 2

Project Understanding



The objective of this project is to implement a web proxy server and a simple cache run on the same machine, using IPC mechanisms to communicate.

In order to achieve this purpose:

- When proxy process receives a request, it will query the cache to see if the file is available in local memory already. Cache process check the presence of requested file and relay it to proxy process. When cache misses, proxy will send a GF_FILE_NOT_FOUND header to client.
- Memory-based IPC was used to transfer file content between proxy and cache process, since it is more efficient than message-based IPC for large data. A shared memory pool was created by the proxy process for multithreaded downloading purpose.
- Message-based IPC was used to send cache request from proxy to cache process, since it is simpler to implement. A message queue object was created by cache process.
- Two semaphores were used for each shared memory segment to synchronize between cache and proxy process: a “read” semaphore to inform the proxy to read from shared memory, a “write” semaphore to inform the cache to write into shared memory.
- I used the POSIX API to implement the IPC mechanisms. Since the interface is simpler and thread safe.
- Since the cache and proxy process may be started in any order, if the cache/proxy cannot connect to the IPC mechanisms, they will delay for a second and try again.
- When received SIGTERM or SIGINT signal, cache/proxy process will firstly remove all IPC objects before terminate, since the objects will stay until system reboot unless been manually cleaned up.

Flow of Control

Proxy:

1. Initialize shared memory pool:
   - Create N shared memory objects.
   - Set the size of each object.
   - Enqueue their name to a queue.
2. Open message queue.
3. Loop forever:
   - Boss thread:
     Receive requests from clients
   - Each worker thread:
     Get a shm object from the pool
     Open and map that object into virtual address space
     Create two semaphore objects

Send cache request via message queue: file path, shm's name, segment size, sem's name
Read file length from shared memory, send header to client
Read file from shared memory and send it to client if file exist
Unmap and close shm object
Close and unlink sem objects.
Put the shm object back to the pool

4. Handle SIGINT or SIGTERM signal:
   - Unlink all shm objects and cleanup the pool
   - Close message queue

Cache:

1. Create message queue object
2. Loop forever:
   - Boss thread:
     Receive requests from proxy
   - Each worker thread:
     Open and map the shm object into virtual address space
     Open the semaphore objects
     Check the existence of requested file and write the file length into shared memory
     Write the file into shared memory if file exist
     Unmap and close shm object
     Close sem objects.
3. Handle SIGINT or SIGTERM signal:
   - Close and unlink message queue object

Testing

- Run proxy process with different thread numbers, segment numbers, segment sizes
- Run cache process with different thread numbers
- Run client process with different thread numbers, request numbers
- Request for nonexistent files
- Request for files with small sizes, large sizes, same sizes, mixed sizes
- Kill/exit proxy process
- Kill/exit cache process

Reference

- https://curl.se/libcurl/c/
- https://man7.org/linux/man-pages/man7/shm_overview.7.html
- https://man7.org/linux/man-pages/man7/mq_overview.7.html
- https://man7.org/linux/man-pages/man7/sem_overview.7.html
- https://www.thegeekstuff.com/2012/03/catch-signals-sample-c-code/