Abstract: 0.25~0.5 pages
This article presents the design of a synchronized multi-threaded file proxy server architecture and evaluation metrics of the implementations of the architecture. Compared with common Web servers, the file server aims to focus on file-related requests and optimize by leveraging the boss-worker pattern and caching file mappings to avoid some unnecessary disk-bound workloads. Moreover, the server can be easily extended for other purposes because it achieves these results using APIs and facilities reusable in many applications including cloud disk, e-library, etc.

Furthermore, the optional optimizations related to the idea of the proxy pattern are discussed as refinement. In the extra considerations, a so-called zero-copy tech is discussed.

Most of the ideas mentioned here originate from my design in some other OMSCS courses and my internship experience at ByteDance. So I would self-reference some key designs of mine I produced before.

Problem Space: ~1 page
The basic mechanism of a typical file server is simple: when the client connects to and sends a request to the server, the server opens the requested file, reads the contents from the file, and writes the chunks of the file continuously to the client via the socket. The server will close the connection when all contents of the file are sent. However, there are three core problems with conventional file servers or the naïve version of the server mentioned here. They involved the performance and the efficiency of the server as well as its limited static file range.

The first core problem that matters is the performance of the file transfer process which can be evaluated by requests being processed per second (rps). Usually, a server would simultaneously receive requests from a large volume of clients. If there is only one thread or process that could handle the requests, only one client could get the feedback (response header and file chunks) from the server while the others would starve. Specifically, for a single process pattern, when the process blocks on a file IO, there are no other similar running processes that could participate in the service. This results in the idle or waste of the multi-core processor of modern computers.

The second problem is that a naïve file server would always try to open a requested file to get a file descriptor and create relevant data structures in the background every time when a request comes in. This step involves some system calls and extra cost on the file system and is supposed to be very inefficient.

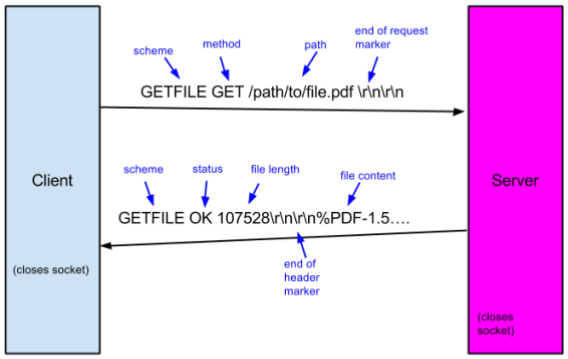
The third apparent problem is that all the files that could be downloaded could only be the files that are statically stored on the server’s disk. A typical scenario of this is that a client fetches a list of files in the current directory via the TCP connection and inputs a command to specify one or a few files to start the downloading procedure. Therefore, the available file range is restricted only to the file list and the client could only request files in the list. Otherwise, the client would get a file-nonexistent prompt. There are no facilities offered to obtain the files from the Internet.

To make full use of the hardware resources of modern machines, improve the file transfer efficiency, and extend the file ranges we could download, we can make an attempt to try to solve the three problems of conventional file servers.

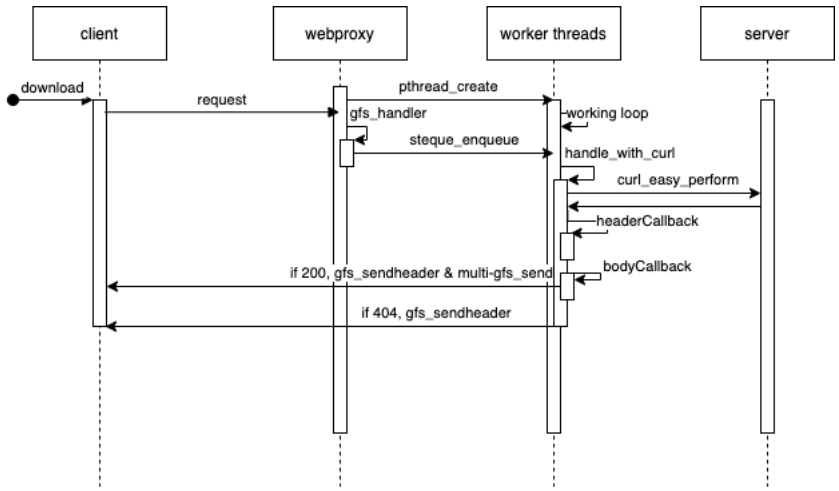
Baseline Design with Need Finding Exercise: ~1.5 pages

In the boss-worker thread pattern, the main thread will continue listening to the socket and accepting new connection requests. Then the requests on these new connections will, however, be fully handled by worker threads. The pool of worker threads should be initialized beforehand with the POSIX thread library to avoid creating costs on the fly. Once the worker threads have been started, they would wait on an empty task queue until the boss thread accepts a connection from a client and enqueue the request as a task into the queue. Once any of the workers get the mutex/lock for the task queue, they could continue to run to finish the task which is read and send file chunks to the requesting client.

An example of requesting the file /path/to/file.pdf could be shown in the following figure.



The control flow can be described by the following more fine-grained sequence diagram. The Web proxy here achieves higher performance by leveraging the boss-worker pattern. As for the second problem – file caching, more details would be given in the Design Refinements section. As for the third problem, to extend the downloadable file range, the web proxy server program would relay the requests and files between the client and a public HTTP server on the Internet (i.e. github.com/path/to/file.pdf). This feature could be implemented with the curl library [1].



My key designs:

1. Use a set of structures including task queue, mutex, and condition variable to synchronize the boss and workers.
2. Continuously call <stdio.h>'s function `char *fgets(char *s, int size, FILE *stream);` to read in characters from stream and store them into the buffer pointed to by s until NULL returned when EOF occurs. Each time the read-in characters would be sent to the client through the write() function.
3. Support both IPv4 and IPv6 [3][4]. "The best approach is to create an IPv6 server socket that can also accept IPv4 connections." To do so, create a regular IPv6 socket, turn *off* the socket option `IPV6_V6ONLY` (which is by default turned on Linux), bind it to the "any" address, and start receiving. IPv4 addresses will be presented as IPv6 addresses, in the IPv4-mapped format."
4. Leverage the [1] to implement the handle_with_curl callback to download files from HTTP servers on the Internet
5. Use "404" as a string needle to match the header status to decide if the file exists on the public HTTP server or not. This adheres to the requirement on message headers of the HTTP protocol.
6. Use "content-length" as a string needle to match the header and retrieve its trailing numbers as an integer and pass it in as part of the responsive header to tell the client the size of the Internet file (i.e. github.com/path/to/file.pdf). This field exists implicitly in many servers on the Internet while some of them offer non-fixed-length file downloads without that field.

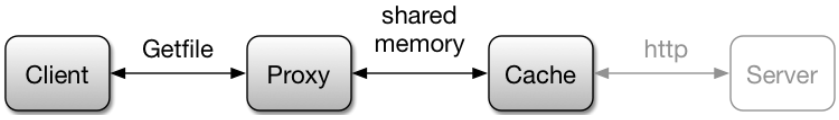
The downloaded file could be checked by comparing their SHA256 (256-bit) checksums by running the Linux routine `sha256sum downloadedFile` and comparing the result with the original file's sha256sum which is transferred along with the file content.

Evaluation metrics for this design would be the number of requests processed per second or the throughput (bytes/second), which can be compared with the basic design.

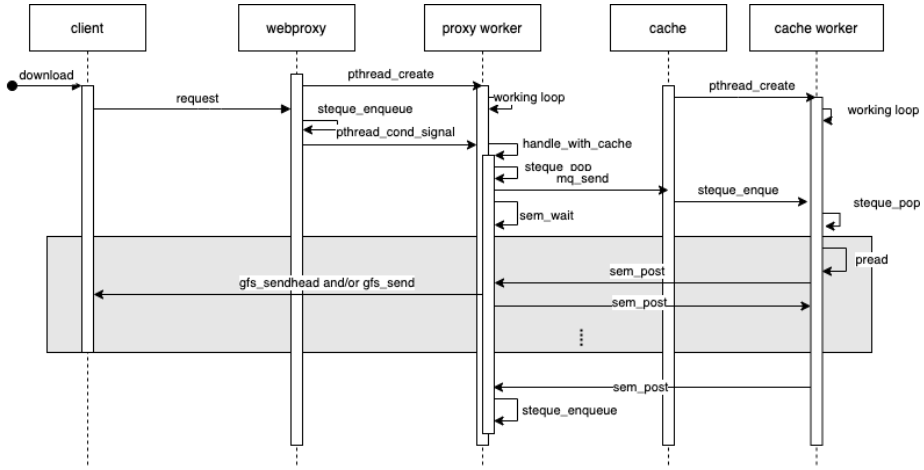
Design Refinements: ~1.5 pages

This part is about a refined design of an extra cache mechanism that will run on the same machine as the Web proxy and communicate with it via shared memory [6] and message queue [7]. Compared with the baseline design, a file caching mechanism is implemented here to tackle the second problem – efficiency.

The mechanism involves two phases. The first phase is providing a file cache on the local cache server for the remote Internet files. The second phase is providing a mapping from the file path to the file descriptor in the cache server process's memory space. The relationships among components could be illustrated as the following. The HTTP server on the Internet (i.e. github.com) is lowlighted because it is optional when the service is deployed on some organization's LAN for internal file downloading.



The control flow can be depicted by a more fine-grained sequence diagram. Again the public HTTP server is omitted again for convenience because it is a relatively external component implemented by a third party.



The shared memory segment is defined as the following:

```
typedef struct Shmbuf { // Buffer in shared memory
    sem_t wsem; // Writer semaphore
    sem_t rsem; // Reader semaphore
    int emptyFile;
    int taskId;
    size_t fileLen; // The file length to be transferred
    size_t cnt; // Number of bytes used in 'buf'
    char buf[]; // Data being transferred
} Shmbuf;
```

My key designs:

1. Shared memory segments are used as a communication medium between the proxy and the cache server.
2. To make them reusable, there is a protected queue to maintain them with mutex and condition variables.
3. The shared memory segments are initialized in the boss thread containing some flags and semaphores besides the data buffer. The reason is that each segment can only be poped, used, and recycled for one request at a certain moment.
4. The semaphores offer the essential synchronization mechanism for the reader-writer pair (the proxy and the cache server) to decide their turns.
5. The flags contained in the shared memory segment involve the handling of file-not-found scenarios, and it is necessary to offer a place to record the file length and chunk length because the proxy needs to get the information from the cache to decide if it received all chunks of the requested file.

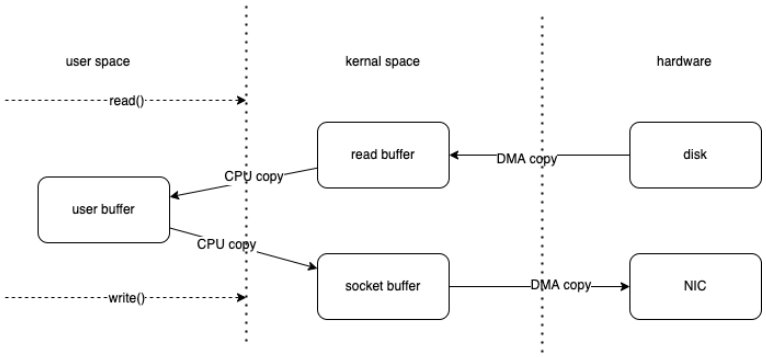
Evaluation metrics for this design would be the number of requests processed per second or the throughput (bytes/second), which can be compared with the basic design. Besides, the file integrity could also be checked by the checksum256 method mentioned in the Base Design section.

Additional Design Considerations: 0.5~1 pages
Although the high-level design seems completed enough there is still some additional optimization that could be achieved by the so-called zero-copy technology.

As we know, in a Web server execution flow, read() is called to read data from the hard disk into the kernel buffer and copy it to the user buffer before write() is called to write the data to the socket buffer, and finally send to the network interface card.

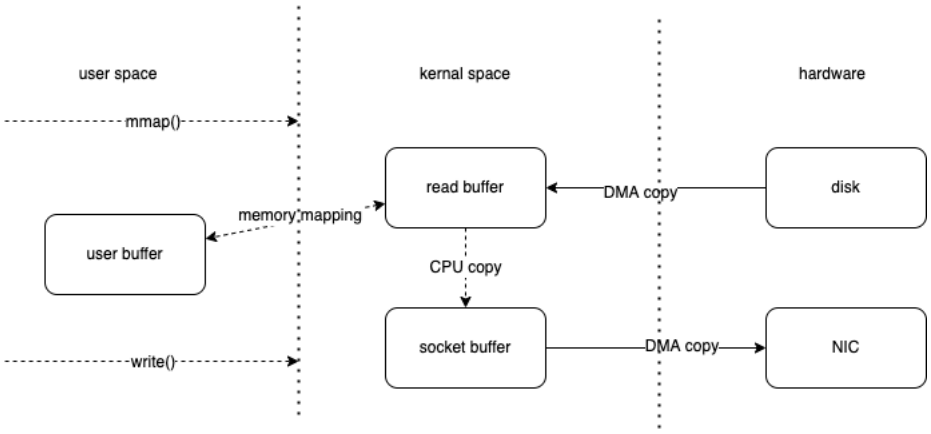
In the whole process, there are 4 user mode and kernel mode context switches and 4 copies. The specific process is as follows:

1. The user process makes a call to the operating system through the read() method and the context changes from the user mode to the kernel mode;
2. The DMA controller copies data from the hard disk to the read buffer;
3. The CPU copies the read buffer data to the application buffer, the context changes from the kernel mode to the user mode, and read() returns;
4. The user process makes the call through the write() method, and the context changes from user mode to kernel mode;
5. The CPU copies data from the application buffer to the socket buffer;
6. The DMA controller copies data from the socket buffer to the network interface card and the context switches from kernel mode back to user mode, and write() returns.

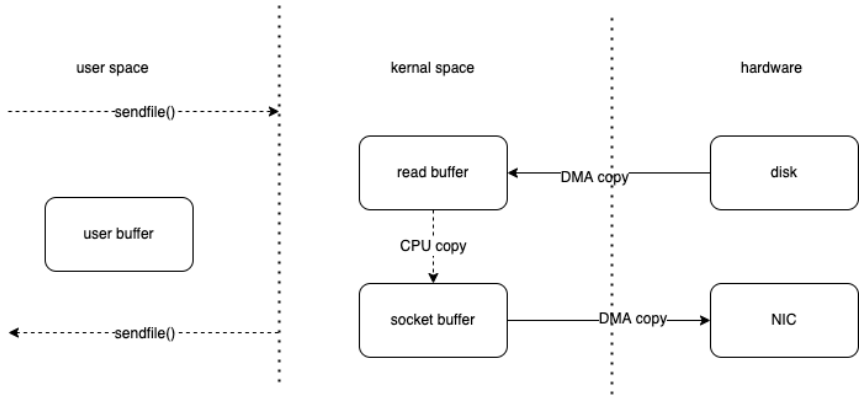


Zero-copy technology [5] means that when a computer performs an operation, the CPU does not need to copy data from one area of memory to another specific area. This technology is usually used to save CPU cycles and memory bandwidth when transferring files over the network. It does not mean that there is really no data copy during the IOs. Its aim is to reduce the user mode and kernel mode switch times and CPU copy times.

One implementation of the zero-copy tech is called mmap+write, which simply replaces the read operation in conventional read+write with mmap() to reduce one copy. The mmap() is a system facility mapping the address of the read buffer to the address of the user buffer, thus reducing the CPU copy times from the read buffer to the user buffer.



The sendfile(), as another implementation, also reduces one CPU copy but reduce two extra context switches compared to mmap(). The sendfile() is a system call function introduced after the kernel version of Linux2.1. By using sendfile(), data can be directly transferred in the kernel space, thus avoiding the copy between the user space and the kernel space. At the same time, because sendfile() is used instead of read+write, a system call time can be saved.
The SendFile method IO data is completely invisible to user space, so it can only be used in cases where user space processing is not required at all, such as the static file server of my design.



Again, the evaluation metrics for this design could be the number of requests processed per second or the throughput (bytes/second), which can be compared with the baseline design and the design refinements. Besides, the file integrity could also be checked by the checksum256 method mentioned in the Baseline Design section.

Reference

1. libcurl's “easy” C interface. http://curl.haxx.se/libcurl/c/
2. Linux manual. https://man7.org/linux/man-pages/man1/man.1.html
3. Beej's Guide to Network Programming. https://beej.us/guide
4. Stack Overflow. https://stackoverflow.com/questions/1618240/how-to-support-both-ipv4-and-ipv6-connections
5. Zero-copy. https://en.wikipedia.org/wiki/Zero-copy
6. Share memory. https://en.wikipedia.org/wiki/Shared_memory
7. Message queue. https://en.wikipedia.org/wiki/Message_queue