Multi-Threaded Web Server with IPC and Semaphores - Design Document

by Luís Correia 125264; Guilherme Martins 125260
Operating Systems | P3

Introduction

This document provides a detailed description of the design and architecture of a high-performance concurrent web server. The project focuses on leveraging both multi-process and multi-threading techniques to efficiently handle a large volume of simultaneous HTTP requests.

The core design centers on a Master-Worker model. The master process is responsible for network listening and accepting new client connections, while a pool of worker processes and their internal threads handle the actual request processing.

Overview

Each parent - master - process listens for and accepts IPv4 TCP connections. When a connection is accepted, a child - worker - process is forked in order to handle the related work. The socket file descriptors are sent to the children by the parent over a UNIX socket pair, using *sendmsg()* and *recvmsg()*.
Each child process then creates a thread pool, whose threads call *workerThread()* and block on *recvmsg()* to receive client file descriptors.
In order to properly control memory usage and coordinate synchronization and queueing of processes, a shared memory region is used to store semaphores and statistics.

Component diagram

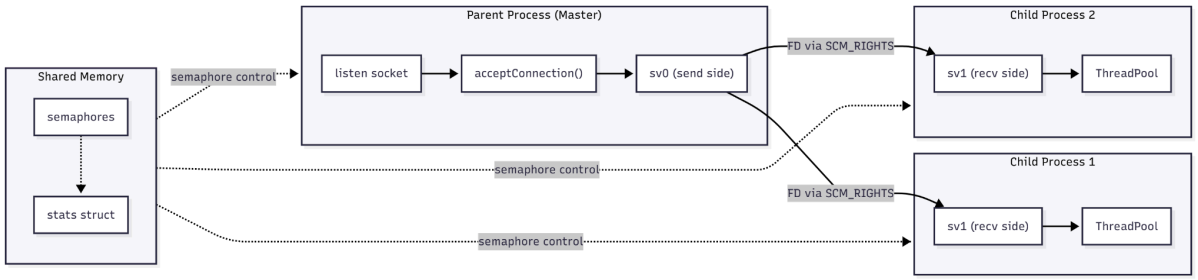

Fig.1: Component diagram of the server.

The following diagram illustrates how connections are accepted and processed by the server, as described previously. In this case, two worker processes are created, each one having their own independent thread pool to handle the connection-related work.

Process and file descriptor sequence diagram

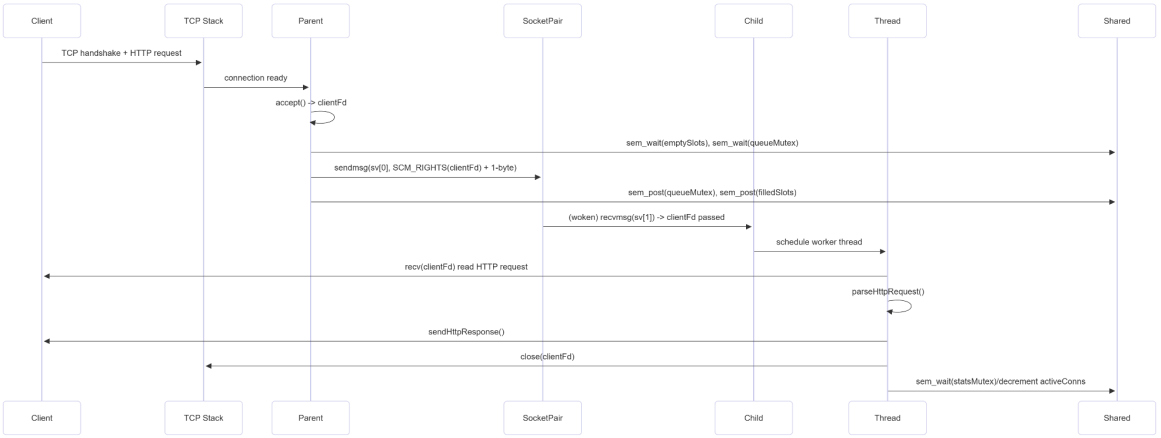

Fig.2: Process and file descriptor-passing sequence diagram of the server.

The following sequence diagram serves to illustrate how processes and file descriptors are passed in the server.
The client and TCP stack first establish a TCP connection via a TCP handshake, with an HTTP request being made. When the connection is ready, the master process then accepts it, and passes the client file descriptor to the socket pair. Afterwards this file descriptor is then received by a worker process via the socket pair, which then schedules a thread from its own thread pool. This thread then reads the HTTP request and parses it, sending an HTTP response back to the client and closing it.

Thread pool sequence diagram

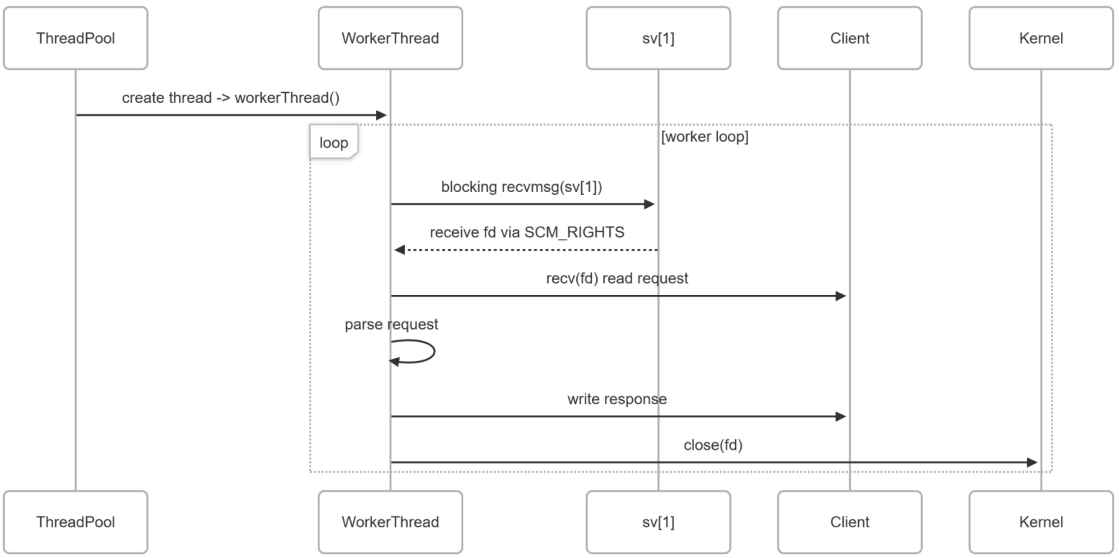

Fig.3: Sequence diagram of the thread pool of a worker process in the server.

When a worker process is created and its thread pool is initialized, multiple threads are created, which each run the *workerThread()* function, waiting for any connection-related work.
Each thread blocks on *recvmsg(sv[1])*, that is, a thread only wakes up when it receives a client file descriptor from the master process via the socket pair. Threads will then parse the client's HTTP request and send it an HTTP response, closing the client file descriptor and going back to waiting for a new one.

Connection acceptance flowchart

Fig.4: Flowchart of how connections are accepted on the server.

The process calls *accept()* to obtain the client file descriptor, checks for errors, and if successful, waits on semaphores to ensure queue capacity. A message is then prepared, which contains the file descriptor, and it's sent over the socket pair to a child process. After sending, the master updates semaphores and shared statistics before returning, completing the connection-handling setup.

Worker thread flowchart

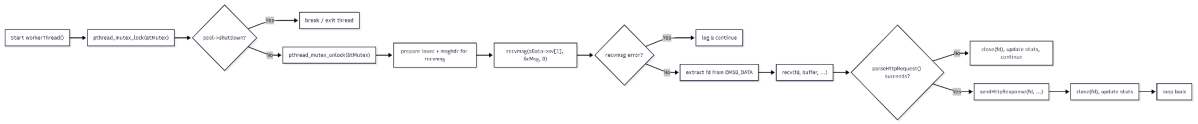

Fig.5: Flowchart detailing how a worker thread functions.

Each thread first locks a mutex to check whether the thread pool is shutting down. If not, it unlocks the mutex and prepares a message buffer to receive a client file descriptor via *recvmsg()* on the socket pair. The HTTP request is then read from the client, parsed, and, if no errors occur, the HTTP response is sent. After handling the request, the thread closes the client connection, updates shared statistics, and loops back to wait for the next file descriptor. Error paths (like *recvmsg* or request parsing failures) are logged, cleaned up, and the thread continues to handle new connections.

Synchronization analysis

Several shared resources require careful synchronization to avoid the emergence of race conditions and to ensure that the server operates properly.
The primary shared resources include the connection queue, managed by the semaphores *emptySlots* and *filledSlots* and protected by *queueMutex*, and the statistics *struct*, which tracks active connections and total requests, being protected by *statsMutex*. Additionally, each child process contains a thread pool with threads that check a shutdown flag protected by *tMutex*.
Semaphores are used for coordination. The parent process - acting as a producer - accepts connections and sends file descriptors to children, while the threads inside child processes - acting as consumers - receive and handle these connections. *queueMutex* ensures that only one thread or process modifies the queue at a time, and *statsMutex* guarantees that updates to the shared statistics are atomic. Worker threads block on *recvmsg()* in order to receive file descriptors from the parent, which naturally synchronizes the handoff of client connections without busy-waiting.
There are several critical sections, such as the parent sending a file descriptor to the child, updating the connection queue and semaphores, and updating statistics after a request is handled.

Conclusion

This design document was able to accurately and concisely describe the design component of the web server, especially key aspects like how the master process and worker processes interact, share memory and are synchronized, how connections are accepted and HTTP requests parsed.

References

- https://pubs.opengroup.org/onlinepubs/7908799/xns/syssocket.h.html
- https://dev.to/jeffreythecoder/how-i-built-a-simple-http-server-from-scratch-using-c-739
- https://pubs.opengroup.org/onlinepubs/009695399/basedefs/netinet/in.h.html
- https://pubs.opengroup.org/onlinepubs/007904875/basedefs/sys/mman.h.html
- https://pubs.opengroup.org/onlinepubs/7908799/xsh/semaphore.h.html
- https://stackoverflow.com/questions/29331938/what-things-are-exactly-happening-when-server-socket-accept-client-sockets
- https://stackoverflow.com/questions/11461106/socketpair-in-c-unix
- https://www.ibm.com/docs/en/ztpf/1.1.2025?topic=apis-socketpair-create-pair-connected-sockets
- https://www.ibm.com/docs/en/i/7.6.0?topic=processes-example-worker-program-used-sendmsg-recvmsg
- https://www.geeksforgeeks.org/c/memset-c-example/
- https://stackoverflow.com/questions/8481138/how-to-use-sendmsg-to-send-a-file-descriptor-via-sockets-between-2-processes
- https://www.quora.com/How-can-you-pass-file-descriptors-between-processes-in-Unix
- https://stackoverflow.com/questions/1788095/descriptor-passing-with-unix-domain-sockets
- https://stackoverflow.com/questions/14721005/connect-socket-operation-on-non-socket
- https://stackoverflow.com/questions/2358684/can-i-share-a-file-descriptor-to-another-process-on-linux-or-are-they-local-to-t
- https://www.ibm.com/docs/en/zos/3.2.0?topic=calls-sendmsg
- https://linux.die.net/man/7/unix
- https://linux.die.net/man/3/cmsg
- https://www.ibm.com/docs/en/zos/2.5.0?topic=functions-sendmsg-send-messages-socket
- https://stackoverflow.com/questions/8359322/how-to-share-semaphores-between-processes-using-shared-memory
- https://mimesniff.spec.whatwg.org/#javascript-mime-type
- https://www.reddit.com/r/learnjavascript/comments/r4ss8t/how_to_parse_data_from_txt_file_using_javascript/
- https://pubs.opengroup.org/onlinepubs/007904975/functions/fread.html
- https://linux.die.net/man/2/send
- https://man7.org/linux/man-pages/man3/getopt.3.html
- https://pubs.opengroup.org/onlinepubs/009696799/functions/getopt.html
- https://serverfault.com/questions/146605/understanding-this-error-apr-socket-recv-connection-reset-by-peer-104