Assignment 2 Report
Internet Programming
VU Amsterdam 2017

Christopher Esterhuyse
student 2553295
cee600

System used for Testing

Type	64-bit Ubuntu
Description	Ubuntu 16.04.3 LTS
Kernal release	4.10.0-33-generic
Processor	Intel Core i7-7500U CPU @ 2.70GHz x 2
Graphics	Chromium
Memory	28.4 GB
openjdk version	1.8.0_131
OpenJDK Runtime Environment	build 1.8.0_131-8u131-b11-2ubuntu1.16.04.3-b11

1 Key-Value Server

1.1 Shared Memory Hash Map Architecture

For all server programs (`serv1` - `serv4`), the same underlying data structure is used to store key-value pairs as necessary between processes and threads in shared memory. The structure is abbreviated in the code and henceforth in the report as `smkv` for 'shared memory key-value'.

1.1.1 Hash Table

At its heart, the `smvk_hash_table` structure contains 32 [1] of entries, each with a corresponding mutex lock. Processes/threads that interact with key-value pairs first determine the relevant table entry by hashing the key [2] , acquire its associated mutex lock, and then complete their work before again unlocking.

1.1.2 Chunk

Each hash table entry *references* a `smkv_chunk` structure in shared memory. These chunks are simply arrays of 16[1] key-value pair slots, along with an integer for bookkeeping. These chunks can be thought of as nodes in a linked list, as they also contain a 'next' field for a reference to a subsequent chunk.

1.1.3 Key-Value Pair

These simple structures contain two 64-byte[1] buffers for null-delimited string representations of keys and values respectively.

[1] All the constants described here are defined in header files. They have been chosen to hopefully cover all reasonable test cases without causing unnecessary waste of memory. They can be changed at will inside the source files `protocol.h` and `hash_map_common.h`

[2] *sdbm* is used as the hash function. See `http://www.cse.yorku.ca/~oz/hash.html`

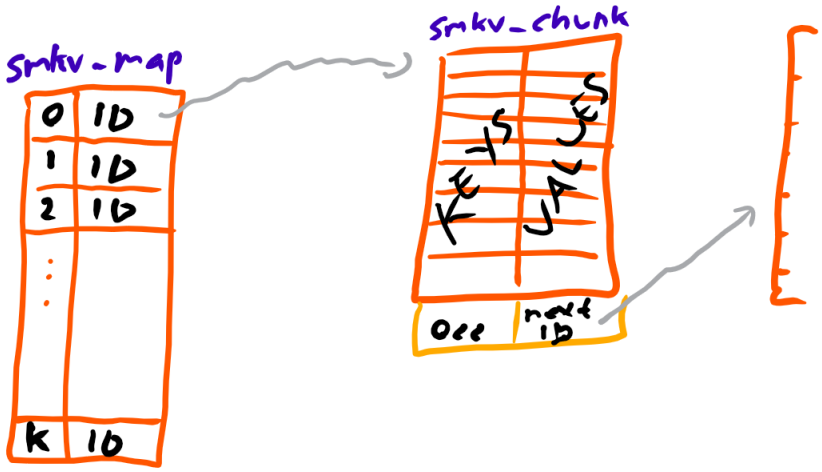

Figure 1: `smkv_hash_table` and `smkv_chunk` structures, each in their own shared memory segment. Pointers between them are indirect, instead referring to the shared memory IDs of the shared memory segments, rather than direct pointers.

1.1.4 Local

All other structures mentioned thus far exist in shared memory, and are thus attached to a process's memory as appropriate. However, the `smkv_local` structure is stored in a process's local memory only. Its purpose it to facilitate translation from shared memory IDs of structures in shared memory, and pointers to their locations in the process's own memory. This applies to chunks, as well as the single hash table itself. Whenever an attempt is made to find the pointer for a shared memory ID that is not contained in the `smkv_local`, the shared memory has not yet been attached. This can then be done lazily. Lastly, the `smkv_local` contains a mutex lock of its own. This lock is used for thread safety if enabled at initialization.

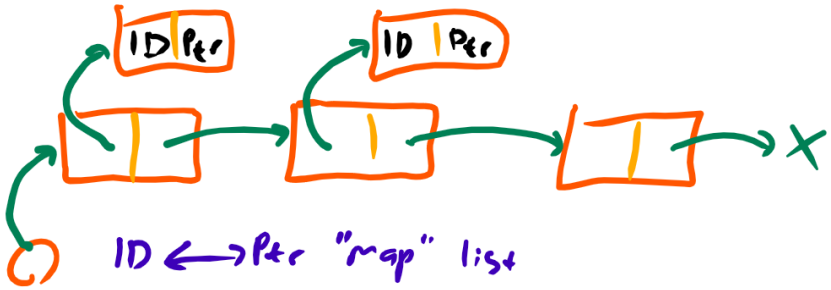

Figure 2: Linked list of the `smkv_local` struct. Each node points to a struct associating a shared-memory ID with a pointer

1.2 Protocol

Client and server communicate via TCP. Their messages have variable lengths, but when messages have been received in their entirety can be deduced from how many null bytes have been received (as they delimit the ends of strings).

1.2.1 client →get→ server

The get request begins with a character 'g' (unicode 103). The subsequent byte begins a null-terminated string for the key. All these bytes including the terminating null are sent on the wire. For instance the request 'get farm' produces a payload of [g, f, a, r, m, \0].

1.2.2 client ← get_response ← server

The response begins with the value for the requested key as a null-terminated string, without any preamble at all. All bytes including the null are sent on the wire. For instance, the response of the value 'cow' will produce a payload of [c, o, w, \0].

1.2.3 client → put → server

The put request begins with a character 'p' (unicode 112). The subsequent byte begins a null-terminated string for the key. Immediately afterwards, the value is included as another null-terminated string. All these bytes including the terminating nulls are sent on the wire. For instance the request 'put a b' produces a payload of [p, a, \0, b, \0]. The server knows the message has arrived in its entirety once two null bytes have been counted.

1.3 Server reading behavior

Clients are able to send multiple requests before terminating the connection. It is also possible that a server will receive only part of a message with each `read()` from the client's socket. To work around this, the server will read from the socket a byte at a time, and operate on the request once a complete request is received. If no bytes are available, the server closes the connection. If more requests are waiting, the server will return to this socket later and `read()` the requests at that time.

1.4 Individual Programs

1.4.1 Client

This program simply attempts to make a TCP connection to the given host name and port. If the name cannot be resolved, the client will also try to interpret the address in 4-dot notation. If this fails, the program exits.

Once connected successfully, the client parses its program arguments (starting at position 3), and prepares a buffer to send packets on the wire to the server (as described in 1.2). Each isolated request `write()`s in isolation, but how it is sent (in terms of actual packets) may differ. In the event of sending a `GET` to the server, the client waits for a server response; once it has been `read()` in entirety, the client prints the response to `STDOUT` and continues to the next request. Once all requests have been send and all responses received, the program exits.

1.4.2 Serv1

This server is sequential, and thus can only service one message at a time. However, since the services it provides can be executed quickly, it is beneficial to avoid the server waiting for all of an individual client's commands. Thus, the server sits in a `select()` loop. For each iteration, the server

1.4.3 Serv2

This server makes use of pop-up processes. The parent process waits in a server loop doing nothing but accepting and forking. Each forked process creates its own `smkv_local` structure and goes to work servicing a single client in a dedicated loop.

1.4.4 Serv3

This server makes use of a predefined pool of pre-forked processes. Each process waits in an accept loop The `accept()` call is protected by a shared-memory mutex lock to prevent swarms of wake-ups between processes sleeping on the `accept()` call. Upon accepting a client, each child process releases the lock and enters a service loop, dedicating itself temporarily to the accepted client until the connection is closed.

1.4.5 Serv4

This server is structurally similar to the pre-forked serv3. Threads are spawned by the server at the beginning. They each begin listening and race one another to accept clients, returning to their server loop when done. The `accept()` call is protected by a mutex between threads for performance reasons.

Creation of this server using the same structures used for serv1-serv3 proved a challenge. Threads a fundamentally different to forked processes, as they share an address space. Thus, threads need not call `shmat()` each for every `smkv_chunk` they want to access. Given the chance, threads would race for attaching the chunk (requiring a lock to avoid duplication and nasty race conditions). With such a lock in place, the `smkv_local` list would become a bottleneck, making it all the more frustrating that the structure really isn't necessary for sharing data within one process, as the hash map doesn't need to be associated with shared memory at all.

For this reason, serv4 has its own version of the `smkv` hash map, aptly named just `kv`, along with appropriate non-shared memory versions of `smkv_chunk`, etc. The difference is that this hash map is allocated in the usual way with `malloc()`, and chunks are linked with regular pointers. Without the extra layer of indirection, the local structure is not needed and threads access the `kv_hash_table` directly. A mutex lock per hash table entry protect the data from race conditions in exactly the same fashion as before.

2 Talk

This program allows two users to chat via a TCP connection. Once it begins, the program goes into either client or server mode depending on program arguments. Once the TCP connection has been established, the program forks. The child process enters a loop responsible for reading from the socket and printing to `STDOUT`. The parent process reads input from `STDIN` one byte at a time, greedily writing to the socket. The connection is closed when either program sends a signal (CTRL+C) to interrupt the program.