Deign Document: ASGN-3 `Adding key-value store to multithreaded HTTP server`

Hang Yuan
CruzID: `hyuan3`

1 Goals

The goal of this project is to implement multi threads HTTP server with key-value store method to improve the performance of server. Key-value store method is implemented like hash function with key and value. The KVS file contains two parts: one stores all the keys including filename, length, and pointer to the data range, the other one stores the file data.

To enhance performance, both client and server cut off message format check since message between client and server is secured to follow the standard HTTP. Except for deleting checking parts, client now will also send Content-Length to server in the header for PUT request.

This implementation doesn't follow the suggested way to implement. Details shown below.

2 Design

There are three parts in the design: (1) key-value store implementation; (2) cache functionality cooperating with key-value implementation; (3) multithreading and concurrency control.

2.1 Key-value store implementation

Key-value store allows server to work on one file instead of accessing multiple files to reduce system execution time on file reading and writing. Key-value store(KVS) file stores all data sent from client and ready for get request to obtain the file data. In this implementation, the KVS doesn't support delete function.

A global variable file-descriptor is used to indicate the KVS file.

As indicated in Goals, KVS file is implemented as Hash-map which has a key to indicate the data location. The KVS file is divided into two parts: (1) storing all the keys in the first part; (2) storing all the file data in the second part.

Keys include three parts: 20-byte (encrypted) httpname, 4-byte length, and 4-byte pointer.

Original httpname is 40-byte HEX char. To convert it into 20-byte `uint8_t` array, use `strtol()` build-in function to covert each two char into one `uint8_t` elemenet. 4-byte length is the total length of this file. 4-byte pointer indicate the beginning point of data located in the file.

There is a delimiter number 22,400,000(28 bytes * 800,000) for key and data parts. Before delimiter is the part of keys, and after it is the part file data.

There are two shared variables `kvs_end` and `kvs_key_end` to control the beginning location of new created key or data. After insertion, `kvs_end` should add file length, and `kvs_key_end` should add 28 bytes.


```
Input      : Array of arguments: -N <thread_number> -c <cache_size> -f <file> <address>:<port_number>
Input             : Array length: arg_count
Shared Variable   : fd_kvs
if argc == 1 then
    fprintf("SET UP FAILED")
    exit(EXIT_FAILURE)
end
obtain -N, -c and -f option by getopt();
if kvs_file exists then
    kvs_init(1)  // initialize 800,000 empty entries in kvs-file
else
    kvs_init(0) // fetch existing entries into cache
end
Set up connection: if error occurs, fprintf(error message) and exit(EXIT_FAILURE)
struct thread_info
    int id, client
    pthread_cond_t busy_lock
end
struct entry
    uint8_t name[20]
    uint32_t pointer
    uint_32_t length;
end
// mutex, semaphore, conditional variable initialization
pthread_t thread[thread_number];
semaphore dispatcher
mutex dlock, kvs_end
conditional variable busy_lock
// working thread initialization
for i to thread_number do
    tinfo[i].id = i
    tinfo[i].client = -1
    pthread_cond_init(&tinfo[i].busy_lock, NULL);
    pthread_create(&thread[i], NULL, processing, &tinfo[i]);
end
for infinite loop do
    accept connection -> cl
    sem_wait(&dispatcher)    // wait if no available thread
    pthread_mutex_lock(&dlock);
```



```
// dispatch work to available working thread
for i = 0 to thread_number do
   if tinfo[i].client == -1 then
      tinfo[i].client = cl;
      pthread_cond_signal(&tinfo[i].busy_lock);
      break;
   end
end
pthread_mutex_unlock(&dlock);
```

Algorithm 1: Server side main program loop

```
Input            : bool new_file
Shared Variable : fd_kvs
struct entry *empty_entry = new entry;
if new_file then
   struct entry *empty_entry = new entry;
   for pointer = 0 to KVS_DELIMITER do
      pwrite(fd_kvs, empty_entry, 28, pointer)
      pointer += 28
   end
else
   for pointer = 0 to KVS_DELIMITER do
      pread(fd_kvs, empty_entry, 28, pointer);
      char obj_name[20];
      memcpy(obj_name, empty_entry->name, 20);
      if strcmp(obj_name, "") == 0 then
         return;
      end
      kvs_entry += 28;
      kvs_map.insert(make_pair(obj_name, pointer));
      pointer += 28
   end
end
```

Algorithm 2: `kvs_init()` fill kvs_file out with empty entries or fetch entries into cache

```
Input  : uint8_t * object_name, char * httpname
for i = 0 to 20 do
    char temp[2];
    temp[0] = httpname[i*2];
    temp[1] = httpname[i*2+1];
    object_name[i] = (uint8_t)strtol(temp, 0, 16);
end
```

Algorithm 3: name_converter() 40-byte HEX to 20-byte uint8_t

```
Input            : void *arg
Shared Variable: fd_kvs
struct thread_info *info = (thread_info *)arg;
for infinite loop do
    // make thread sleep until get signal to wake up
    pthread_mutex_lock(&dlock);
        if info->client == -1 then
            pthread_cond_wait(&info->busy_lock, &dlock);
        end
    pthread_mutex_unlock(&dlock);
    // PUT request
    if acition_code == s then
        name_converter(object_name, httpname)
        offset = kvinfo(object_name, file_size);
        write_result = pwrite(fd_kvs, file_buf, recv_result, offset);
        count += write_result;
        offset += write_result;
        while file_size > count and recv_result != 0 do
            memset(file_buf, 0, sizeof(file_buf));
            recv_result = recv(cl, file_buf, sizeof(file_buf) - 1, 0);
            write_result = pwrite(fd_kvs, file_buf, recv_result, offset);
            count += write_result;
            offset += write_result;
        end
        send(cl, "HTTP/1.1 200 OK\r\n\r\n", 19, 0);
    // GET request
    else if action_code == r then
        name_converter(object_name, httpname);
        if (file_size = kvinfo(object_name, -1)) == -2 then
            Send(cl, "HTTP/1.1 404 Not Found\r\n\r\n", 26, 0)
        else
```



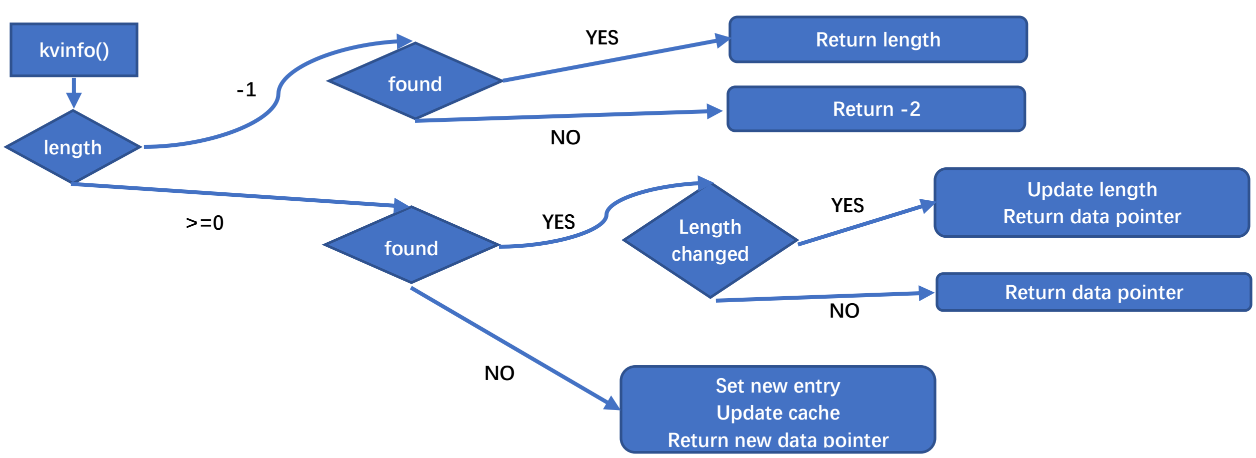
```
            send(HTTP/1.1 200 OK\r\nContent-Length: file_size\r\n\r\n)
            Offset = kvinfo(object_name, file_size)
            if file_size < 4096 then
               read_result = pread(fd_kvs, file_buf, file_size, offset);
               send(cl, file_buf, read_result, 0)
            else
               size_t read_amount = 4096
               while count < file_size do
                  read_result = pread(fd_kvs, file_buf, read_amount, offset);
                  send(cl, file_buf, read_result, 0)
                  offset += read_result
                  count += read_result
                  if (file_size - count) < 4096 then
                     read_amount = file_size - count
                  end
               end
            end
         else
            send(cl, "HTTP/1.1 400 Bad Request\r\n\r\n", 28, 0);
         end
      // reset working thread state
      info->client = -1;
      sem_post(&dispatcher);
```

Algorithm 4: working thread process

2.2 Cache functionality cooperating with key-value implementation

Cache is adopted to store all entries location in the KVS file, trying to reduce the time for GET request instead of reading KVS file. The cache function mainly cooperates with the `kvinfo()` function which will set up entry and data pointer.

`kvinfo()` is the essential function in this project. For basic flow chart of `kvinfo()`, see below:




```
Input            : uint8_t * object_name, ssize_t length
Shared Variables : kvs_end, kvs_entry_end, &kvs_end_lock
struct entry *empty_entry = new entry;
map<string, uint32_t>::iterator iter;
memcpy(obj_name, object_name, 20);
if (iter = kvs_map.find(obj_name)) != kvs_map.end() then
    found = 1;
end
if length == -1 then // return object length
    if found == 0 then // if not found, return -2
        return -2;
    end
    // if found, return length
    uint32_t pointer = iter->second;
    pread(fd_kvs, empty_entry, 28, pointer);
    return empty_entry->length;
else // set new object length
    if found then // if found, update length and return new pointer
        uint32_t pointer = iter->second;
        pread(fd_kvs, empty_entry, 28, pointer);
        if empty_entry->length == length then
            return empty_entry->pointer;
        else
            empty_entry->length = length;
            pthread_mutex_lock(&kvs_end_lock);
            empty_entry->pointer = kvs_end;
            kvs_end += length;
            pwrite(fd_kvs, empty_entry, 28, pointer);
            pthread_mutex_unlock(&kvs_end_lock);
            return empty_entry->pointer;
        end
    else // if not found, create entry, update cache, and return new pointer
        memcpy(empty_entry->name, object_name, 20);
        empty_entry->length = length;
        pthread_mutex_lock(&kvs_end_lock);
        empty_entry->pointer = kvs_end;
        kvs_end += length;
        pwrite(fd_kvs, empty_entry, 28, kvs_entry);
        kvs_map.insert(make_pair(obj_name, kvs_entry));
        kvs_entry += 28;
```



```
        pthread_mutex_unlock(&kvs_end_lock);
        return empty_entry->pointer;
    end
```

Algorithm 5: `kvinfo()` to obtain length or set up entry

No synchronization technics are discussed above, more details in next part.

2.3 Multithreading and concurrency control

For multithreading, this project keeps the basic mutex, semaphore and conditional variable structure in ASGN 2. The mutex for reset(sleep) working thread is moved. Except for this, the implementation introduces three new shared variables: fd_kvs, kvs_end and kvs_entry_end.

For fd_kvs, the file descriptor of KVS file, because it's never changed, there is no concurrency issue. For kvs_end, the end point of KVS file where to add new data, we add a mutex lock to ensure that only one thread can access this shared variable and increment its value. The same reason and resolution for kvs_entry_end, which indicate the end of valid entry. Because once the entry in the KVS file and cache obtains the data pointer, no other working threads can change the range, no concurrency issue there.

