Design Document: adding key-value store to multithreaded HTTP server

Yujia Li
CruzID: yli302

1. Goals

This is a simple multi-threading HTTP server and a simple HTTP client which can only have PUT and GET request. Key-value store is a simple file system to store file block object and data block in a single fvs file. This implementation will avoid open file many times, and improve the performance of server.

2. Changes in client and server

2.1 Client

2.1.1 Implement Content-length in PUT request.

1. Open file check file exist and permission.
   ```
   int32_t fd = open(filename);
   ```
2. Get Content-length of file by fstat.
   ```
   Struct stat sb;
   fstat(fd, &sb);
   size_t content-length = sb.st_size;
   ```
3. Send header
   compose headerbuffer into correct header format.
   Header format:

// PUT 12345678abcdef... HTTP/1.1\r\n Content-Length: #number\r\n\r\n

   ```
   char * headerbuf = parseHeaderbuffer();
   send(headerbuf);
   ```
4. Send data
   No change. Same as asg1 and asg2

2.2 Server

2.2.1 System design

The server will only open file once when create or open fvs file, and store the file descriptor fd to global filed. Read and write will be replaced by kvread and kvwrite. The kvsread and the kvswrite will read and write file object to kvs file. (details about kvs see part 3 and part 4)

2.2.2 Global variable

Mutex and semaphore: for concurrency, same as asg2.

`int fd` is the file descriptor of kvs file.

`uint32_t curr_fvs_end` is a pointer which points to end of kvs file.

2.2.3 Fvs file setup

Flag [-f filename] specifies the filename of kvs file. User must use -f to specify kvs file filename. If no -f flag detected, print no filename error.

After get filename, process following work:

```
fd = open(filename, O_RDWR | O_CREAT);
// init_fvs_file will set up fvs file, and let
// curr_fvs_end point to file end.
init_fvs_file();
```

See detail of ini_fvs_file() function in 3.2.

2.2.4 PUT

Delete this because kvs don't have this problem:
403 error when write to httpfile(write permission deny)

Change pwrite to kvwrite.

```
// kvinfo return 1 if create a new file entry
isNewfile = kvinfo(object_name, content_length);
// receive data from client and write to kvs file
offset = 0
count = recv(data);
while true then
  kvwrite(object_name, count, offset, data);
  offset += count;
  if (offset == content_length) then break;
  count = recv(data);
end
```

2.2.5 GET

Delete this because kvs don't have this problem:
403 error when read from httpfile(read permission deny, read from a directory)

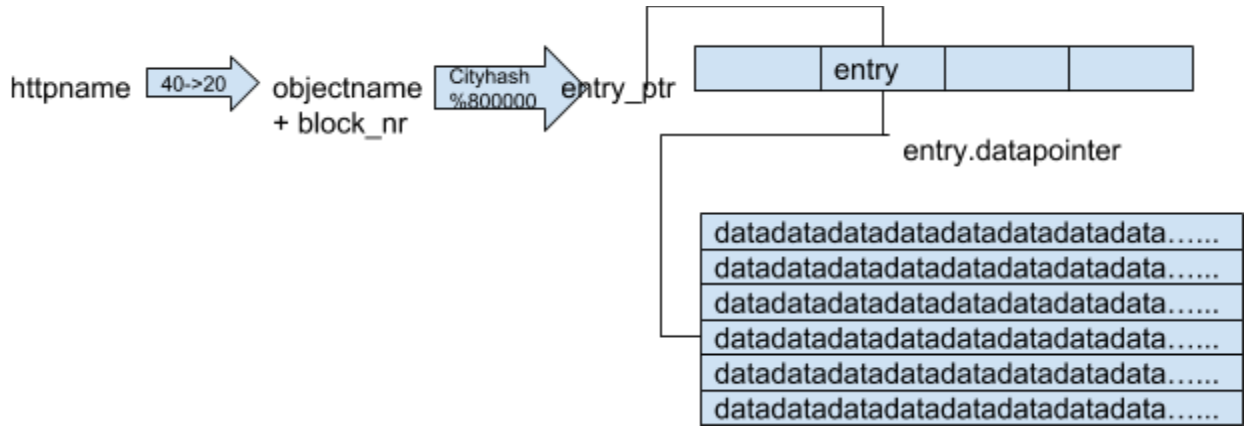
```
// kvinfo return -2 if doesn't find file entry
if ((content_length=kvinfo(name, -1))==-2) then
     error(404 Not Found);
     continue;
end
// receive data from kvs file and send to client
offset = 0;
count=kvread(object_name,sizeof data,offset,data);
while true then
  send(data);
  offset += count;
  if (offset == content_length) then break;
  count=
     kvread(object_name,sizeof data,offset,data);
end
```

2.2.6 Concurrency

Lock when change curr_fvs_end and file object entry.

3. Key-value store system design

3.1 Top level design



kvs file can have max 800,000 entries and 800,000 data blocks.

3.1.1 File object entry

A struct of file object, include: `filename, datapointer, datalength, and block_nr.`

`filename` is a 20 byte buffer of httpname. Convert from 40 byte hex character buffer. (see 4.2)
`datapointer` is a pointer points to data block.
`datalength` if different for block 0 and other blocks. If this entry is block 0, datalength will be the Content Length of whole file. If this entry is not block 0, datalength will be sizeof data buffer in the data block.
`block_nr` means the nth block of this file. For example, if content length of file is 10000, there will be 3 blocks of this file: block 0, block 1, and block 2. And block_nr for these blocks are 0, 1, and 2. To get `block_nr`, use offset / 4096.

3.1.2 Data block

The places where really store data. To find correct place of desired data, use `datapointer + offset_of_this_block`. (see 4.2)

3.1.3 linear probing

Collision of file object entry can happen. Therefore, linear probing is necessary for handle collision.

3.1.3.1 Insert

When collison happen, plus `entry_ptr` by 32(sizeof entry), until find an empty entry. Then insert to this empty entry.

3.1.3.2 Find

After get entry by `entry_ptr`, compare filename and block_nr. If different, plus `entry_ptr` by 32, until find an entry contains same information, or find an empty one.

3.2 initial kvs file

If the kvs file exists, keep track of the pointer to kvs file end, and return 0 to notice this kvs file exists. If the kvs file doesn't exist, create new one, and return 1 to notice this kvs file has been created.

To create kvs file, set 25,600,000 (get by 32 * 800,000) byte as empty entries. See more detail on 4.2.

3.3 kv functions

3.3.1 kvwrite

The way to use `kvwrite` is the same as `pwrite`. `kvwrite` updates file object entry and write data to data block based on offset. See more detail on 4.2.

3.3.2 kvread

The way to use `kvread` is the same as `pread`. `kvread` read data from data block based of offset.

3.3.3 kvinfo

`kvinfo` return necessary information to process `kvwrite` and `kvread` when process PUT and GET request. See more detail on 4.2.

4. Key-value store data structure & algorithm

4.1 data structure

```
// see 3.1.1 for explain of file_object_entry
struct file_object_entry{
  uint8_t httpname[20];
  uint32_t datapointer;
  int32_t datalength;
  uint32_t blocknr;
}
// key for cityhash
struct key {
  uint8_t httpname[20];
  uint32_t blocknr;
}
```

4.2 algorithm

Initial kvs file

```
// if kvs file exists, keep track of pointer to file
end, return 0 to notice exist.
// if kvs file doesn't exist, create new one, return 1
// to notice created.
int32_t init_fvs_file() {
  file_object_entry init_entry;
  offset = 0;
  curr_fvs_end=
     ((((file_sz-25600000)/4096)+1)*4096)+25600000;
  if (file_sz == 0) {
    for (int i = 0; i < 800000; ++i) {
      pwrite(fd, &init_entry, 32, offset);
```

```
      offset += 32;
    }
    curr_fvs_end = file_sz;
    return 1;
  }
  return 0;
}
```

Cityhash

```
uint32_t hash(const uint8_t *object_name, uint32_t
block_nr) {
  struct key key;
  key.httpname = object_name;
  key.block_nr = block_nr;
  return CityHash32(key, 24) % 800000;
}
```

Convert 40 bytes hex char to 20 bytes ascii

```
void hex_to_ascii(uint8_t *object_name, const char
*httpname) {
  for (i = 0; i < 20; ++i) {
    Char * temp;
    temp[0] = httpname[i * 2];
    temp[1] = httpname[i * 2 + 1];
    uint8_t ascii_value = convert this two hex char to
                            one ascii char;
    object_name[i] = ascii_value;
  }
}
```

kvwrite

```
ssize_t kvwrite(const uint8_t *object_name, size_t
length, size_t offset, const uint8_t *data) {
  block_nr = offset / 4096;
  data_offset = offset % 4096;
  entry_ptr = hash(object_name, block_nr);
  // loop for linear probing, iterate entil empty or
find object.
  for (;;) {
```

```
    if (pread(fd, &entry, 32, entry_ptr * 32) == -1) {
      Read error;
      return -1;
    }
    // if object doesn't exist, create entry object and
update data.
    if (object doesn't exist) {
      entry.block_nr = block_nr;
      entry.length = length;
      entry.datapointer = curr_fvs_end;
      curr_fvs_end += 4096;
      if(pwrite(fd, &entry, 32, entry_ptr * 32) == -1){
        Write error;
        return -1;
      }
      return pwrite(fd, data, length, entry.datapointer
                      + data_offset);
    }
    // if this entry contains correct object, write data
    if (entry.httpname == object_name &&
        entry.block_nr == block_nr) {
      // if this is not block 0, update length.
      //  if write a new data, overwrite the Length
      //  if continue write data in a block, plus this
      //   length
      if (block_nr != 0) {
        if (data_offset == 0) {
          entry.length = length;
        } else {
          entry.length = data_offset + length;
        }
      }
      if(pwrite(fd, &entry, 32, entry_ptr * 32) == -1){
        Write error;
        return -1;
      }
```

```
      return pwrite(fd, data, length, entry.datapointer
                        + data_offset);
    }
    // if this entry is not the correct object or
    // empty, find next one
    entry_ptr += 1 * 32;
  }
  return -1;
}
```

kvread

```
ssize_t kvread(const uint8_t *object_name, size_t
length, size_t offset, uint8_t *data) {
  block_nr = offset / 4096;
  entry_ptr = Hash(object_name, block_nr);
  // loop for linear probing, iterate entil empty or
find object.
  for (;;) {
    if (pread(fd, &entry, 32, entry_ptr * 32) == -1) {
      Read error
      return -1;
    }
    // if object doesn't exists, return -2.
    if (object doesn't exists) == 0) {
      return -2;
    }
    // if this entry contains correct object, read data
    if (entry.httpname == object_name &&
        entry.block_nr == block_nr) {
      // if this is block 0, and length greater than
      // 4096, return 4096
      if (block_nr == 0 && entry.length > 4096) {
        pread(fd, data, 4096, entry.datapointer);
        return length;
      } else {
        pread(fd, data, entry.length,
```

```
                    entry.datapointer);
          return entry.length;
        }
      }
      // if this entry is not the correct object or empty,
find next one
      entry_ptr += 1 * 32;
    }
    return -2;
}
```

kvinfo

```
ssize_t kvinfo(const uint8_t *object_name, ssize_t
length) {
  entry_ptr = hash(object_name, 0);
  // for get length
  if (length == -1) {
    // loop for linear probing, iterate entil empty or
    // find object.
    for (;;) {
      if(pread(fd, &entry, 32, entry_ptr * 32) == -1) {
        Read error
        return -1;
      }
      // if object doesn't exist, return -2;
      if (object doesn't exist) {
        return -2;
      }
      // if this entry contains correct object, return
      // its length
      if (entry.httpname == object_name) {
        return entry.length;
      }
      // if this entry is not the correct object, find
      // next one
      entry_ptr += 1 * 32;
    }
```

```
}
// for set length
if (length >= 0) {
  // loop for linear probing, iterate entil empty or
  // find object.
  for (;;) {
    if(pread(fd, &entry, 32, entry_ptr * 32) == -1) {
      Read error;
      return -1;
    }
    // if object doesn't exist, create it and return
    // 1.
    if (object doesn't exist) {
      entry.block_nr = 0;
      entry.length = length;
      entry.datapointer = curr_fvs_end;
      curr_fvs_end += 4096;
      if(pwrite(fd, &entry, 32, entry_ptr * 32)==-1){
        Write error;
        return -1;
      }
      return 1;
    }
    // if this entry contain this object, update
    // length and return 0
    if (entry.httpname == object_name) {
      entry.length = length;
      if(pwrite(fd, &entry, 32, entry_ptr * 32)==-1){
        Write error;
        return -1;
      }
      return 0;
    }
    // if this entry is not the correct object, find
    // next one
    entry_ptr += 1 * 32;
```

```
    }
  }
  return -2;
}
```