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.

1. **Input :** Array of arguments: **-N <thread\_number> -c <cache\_size> -f <file>** **<address>:<port\_number>**
2. **Input :** Array length: **arg\_count**
3. Shared Variable **: fd\_kvs**
4. **if** argc == 1 **then**
5. fprintf(“SET UP FAILED”)
6. **exit(EXIT\_FAILURE)**
7. **end**
8. obtain -N, -c and -f option by getopt();
9. **if** kvs\_file exists **then**
10. kvs\_init(1) // initialize 800,000 empty entries in kvs-file
11. **else**
12. kvs\_init(0) // fetch existing entries into cache
13. **end**
14. Set up connection: if error occurs, fprintf(error message) and exit(EXIT\_FAILURE)
15. **struct** thread\_info
16. int id, client
17. pthread\_cond\_t busy\_lock
18. **end**
19. **struct** entry
20. uint8\_t name[20]
21. uint32\_t pointer
22. uint\_32\_t length;
23. **end**
24. // mutex, semaphore, conditional variable initialization
25. pthread\_t thread[thread\_number];
26. semaphore dispatcher
27. mutex dlock, kvs\_end
28. conditional variable busy\_lock
29. // working thread initialization
30. **for** i to thread\_number **do**
31. tinfo[i].id = i
32. tinfo[i].client = -1
33. pthread\_cond\_init(&tinfo[i].busy\_lock, NULL);
34. pthread\_create(&thread[i], NULL, processing, &tinfo[i]);
35. **end**
36. **for** infinite loop **do**
37. accept connection -> cl
38. sem\_wait(&dispatcher) // wait if no available thread
39. pthread\_mutex\_lock(&dlock);
40. // dispatch work to available working thread
41. **for** i = 0 to thread\_number **do**
42. **if** tinfo[i].client == -1 **then**
43. tinfo[i].client = cl;
44. pthread\_cond\_signal(&tinfo[i].busy\_lock);
45. **break**;
46. **end**
47. **end**
48. pthread\_mutex\_unlock(&dlock);

Algorithm 1: Server side main program loop

**Input :** **bool** new\_file

**Shared Variable** **:** fd\_kvs

1. struct entry \*empty\_entry = new entry;
2. **if** new\_file **then**
3. struct entry \*empty\_entry = new entry;
4. **for** pointer = 0 to KVS\_DELIMITER **do**
5. pwrite(fd\_kvs, empty\_entry, 28, pointer)
6. pointer += 28
7. **end**
8. **else**
9. **for** pointer = 0 to KVS\_DELIMITER **do**
10. pread(fd\_kvs, empty\_entry, 28, pointer);
11. char obj\_name[20];
12. memcpy(obj\_name, empty\_entry->name, 20);
13. **if** strcmp(obj\_name, "") == 0 **then**
14. return;
15. **end**
16. kvs\_entry += 28;
17. kvs\_map.insert(make\_pair(obj\_name, pointer));
18. pointer += 28
19. **end**
20. **end**

Algorithm 2: kvs\_init() fill kvs\_file out with empty entries or fetch entries into cache

**Input :** uint8\_t \* object\_name, char \* httpname

1. for i = 0 to 20 do
2. char temp[2];
3. temp[0] = httpname[i\*2];
4. temp[1] = httpname[i\*2+1];
5. object\_name[i] = (uint8\_t)strtol(temp, 0, 16);
6. end

Algorithm 3: name\_converter() 40-byte HEX to 20-byte uint8\_t

**Input :** **void** \*arg

**Shared** **Variable:** fd\_kvs

1. struct thread\_info \*info = (thread\_info \*)arg;
2. **for** infinite loop **do**
3. // make thread sleep until get signal to wake up
4. pthread\_mutex\_lock(&dlock);
5. **if** info->client == -1 **then**
6. pthread\_cond\_wait(&info->busy\_lock, &dlock);
7. **end**
8. pthread\_mutex\_unlock(&dlock);
9. // PUT request
10. **if** acition\_code == s **then**
11. name\_converter(object\_name, httpname)
12. offset = kvinfo(object\_name, file\_size);
13. write\_result = pwrite(fd\_kvs, file\_buf, recv\_result, offset);
14. count += write\_result;
15. offset += write\_result;
16. **while** file\_size > count **and** recv\_result != 0 do
17. memset(file\_buf, 0, sizeof(file\_buf));
18. recv\_result = recv(cl, file\_buf, sizeof(file\_buf) - 1, 0);
19. write\_result = pwrite(fd\_kvs, file\_buf, recv\_result, offset);
20. count += write\_result;
21. offset += write\_result;
22. **end**
23. send(cl, "HTTP/1.1 200 OK\r\n\r\n", 19, 0);
24. // GET request
25. **else** **if** action\_code == r **then**
26. name\_converter(object\_name, httpname);
27. **if** (file\_size = kvinfo(object\_name, -1)) == -2 **then**
28. Send(cl, "HTTP/1.1 404 Not Found\r\n\r\n", 26, 0)
29. **else**
30. send(HTTP/1.1 200 OK\r\nContent-Length: file\_size\r\n\r\n)
31. Offset = kvinfo(object\_name, file\_size)
32. **if** file\_size < 4096 **then**
33. read\_result = pread(fd\_kvs, file\_buf, file\_size, offset);
34. send(cl, file\_buf, read\_result, 0)
35. **else**
36. size\_t read\_amount = 4096
37. **while** count < file\_size **do**
38. read\_result = pread(fd\_kvs, file\_buf, read\_amount, offset);
39. send(cl, file\_buf, read\_result, 0)
40. offset += read\_result
41. count += read\_result
42. **if** (file\_size - count) < 4096 **then**
43. read\_amount = file\_size – count
44. **end**
45. **end**
46. **end**
47. **else**
48. send(cl, "HTTP/1.1 400 Bad Request\r\n\r\n", 28, 0);
49. **end**
50. // reset working thread state
51. info->client = -1;
52. 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:

**YES**

**Return length**

**-1**

**found**

**kvinfo()**

**Return -2**

**NO**

**length**

**YES**

**Update length**

**Return data pointer**

**Length changed**

**YES**

**>=0**

**found**

**NO**

**Return data pointer**

**NO**

**Set new entry**

**Update cache**

**Return new data pointer**

1. **Input : uint8\_t \*** object\_name, **ssize\_t** length
2. **Shared Variables :** kvs\_end, kvs\_entry\_end, &kvs\_end\_lock
3. struct entry \*empty\_entry = new entry;
4. map<string, uint32\_t>::iterator iter;
5. memcpy(obj\_name, object\_name, 20);
6. **if** (iter = kvs\_map.find(obj\_name)) != kvs\_map.end() **then**
7. found = 1;
8. **end**
9. **if** length == -1 **then** // return object length
10. **if** found == 0 **then** // if not found, return -2
11. **return** -2;
12. **end**
13. // if found, return length
14. uint32\_t pointer = iter->second;
15. pread(fd\_kvs, empty\_entry, 28, pointer);
16. **return** empty\_entry->length;
17. **else** // set new object length
18. **if** found **then** // if found, update length and return new pointer
19. uint32\_t pointer = iter->second;
20. pread(fd\_kvs, empty\_entry, 28, pointer);
21. **if** empty\_entry->length == length **then**
22. **return** empty\_entry->pointer;
23. **else**
24. empty\_entry->length = length;
25. pthread\_mutex\_lock(&kvs\_end\_lock);
26. empty\_entry->pointer = kvs\_end;
27. kvs\_end += length;
28. pwrite(fd\_kvs, empty\_entry, 28, pointer);
29. pthread\_mutex\_unlock(&kvs\_end\_lock);
30. **return** empty\_entry->pointer;
31. **end**
32. **else** // if not found, create entry, update cache, and return new pointer
33. memcpy(empty\_entry->name, object\_name, 20);
34. empty\_entry->length = length;
35. pthread\_mutex\_lock(&kvs\_end\_lock);
36. empty\_entry->pointer = kvs\_end;
37. kvs\_end += length;
38. pwrite(fd\_kvs, empty\_entry, 28, kvs\_entry);
39. kvs\_map.insert(make\_pair(obj\_name, kvs\_entry));
40. kvs\_entry += 28;
41. pthread\_mutex\_unlock(&kvs\_end\_lock);
42. **return** empty\_entry->pointer;
43. **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.