Deign Document: ASGN-2 `Multi-threaded HTTP Server with In-memory Caching`

Hang Yuan
CruzID: `hyuan3`

1 Goals

The goal of this project is to implement multi threads and caching features to enhance the performance of server. This project is developed based on ASGN1's server code. The server should handle multiple request sent from the client simultaneously instead of dealing with requests one by one. In addition, in-memory caching is introduced to allow the server to handle request with data obtained from the cache rather than disk if there is a copy stored in it.

No change in client side but there are a few enhancements to the performance of handling request by replacing address computation to obtain specific arguments with C++ build-in functions. For server side, except for two new features added, the same enhancement was introduced in server side.

The number of thread and the number of caching blocks can be specified by input arguments. Both client and server contain the code copied from the ASGN 1.

2 Design

There are two parts to the design. One is handling multi-thread and one is cache implementation. The server uses POSIX pthread library to implement multi-thread and thread safety. And map and list to implement cache blocks.

2.1 Multi-thread implementation and synchronization control

The server should handle multiple client transactions at one time to improve the throughput of server functionality. To do this, server should split the processing function, which is original server actions for PUT and GET from the main function.

Set up a dispatcher to wake up available working thread and pass thread info to that working thread. The dispatcher will check thread state by checking cl, the file descriptor of accept(). If accept() occurs, set cl and wake up one thread in the thread pool. Before the working thread is waken up, that thread will be in asleep, locking by a conditional variable to wait for signaling up. For the processing function, we set it to be an infinite loop to handle request with a conditional variable waiting for signal. At the end of the function, reset the conditional variable.

In order to avoid data race, every single change to thread state should use mutex to create a critical region to maintain atomicity. Use semaphore for writing and mutex for rw_lock to make sure following works:

When there is a reading assignment	Cannot write but can read
When there is a writing assignment	Cannot write and read

```
Input    : Array of arguments: -N <thread_number> -c <cache_size> <address>:<port_number>
Input    : Array length: arg_count
if argc == 1 then
    fprintf("SET UP FAILED")
    exit(EXIT_FAILURE)
end
obtain -N and -c option by getopt();
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
// mutex, semaphore, conditional variable initialization
pthread_t thread[thread_number];
semaphore dispatcher, writing
mutex dlock, rwlock
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   : void *arg
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);
    if acition_code == s then
        // writing semaphore lock
        sem_wait(&writing)
        PUT(httpname, filename)
        sem_post(&writing)
    else if action_code == r then
        // change rw-lock
        pthread_mutex_lock(&rwlock);
        nreaders += 1;
        if nreaders == 1 then
            sem_wait(&writing);
        end
        pthread_mutex_unlock(&rwlock);
        GET(httpname, filename)
        // reset rw-lock
        pthread_mutex_lock(&rwlock);
        nreaders -= 1;
        if nreaders == 0 then
            sem_post(&writing);
        end
        pthread_mutex_unlock(&rwlock);
    end
    // reset working thread state
    pthread_mutex_lock(&dlock);
    info->client = -1;
    sem_post(&dispatcher);
    pthread_mutex_unlock(&dlock);
```

Algorithm 2: working thread process

2.2 Cache implementation

The server adopts in-memory cache to improve the performance to access the file that have been accessed. The build-in functions of C++'s map and list are adopted to achieve this implementation goal.

The entire cache was formed by a list which is 4KB large for each block. The blocks are assigned to different httpnames stored in map where the pair of httpname and block number is stored. The cache adopts LRU method to maintain the freshest data stored in the cache and make old data be flushed out.

GET:

In GET request execution, the server will first find the httpname in cache_map to check if there is the data stored. If httpname presents in cache, just read from cache and send it out to the client. If the data not exists, the cache will be responsible to retrieve the data from disk and update the data into the cache. Then the server will retry to obtain the data from the cache and do the same thing as the data presents in the cache.

Once the file data is fetched from the cache, the relevant block of that file with the same httpname should be put at the beginning of the cache blocks. In this case, no data blocks will be kicked out.

PUT:

In PUT request execution, the execution is much more complex. When we execute a PUT request, we need first to check if there is data file stored in cache with the same httpname. If the data is found, drop all the relevant block in cache and relevant records in map. Then write the data to the disk and update the file data in the cache.

Because of LRU cache, we will drop the last recently block we use if the entire cache exceeds the max size with new data block. We find the last block in cache, delete the relevant key in the map. Thus, somehow, the data is marked unavailable.

In terms of concurrency, the PUT and GET should be in critical region to avoid data race.

The basic flow chart as followed:

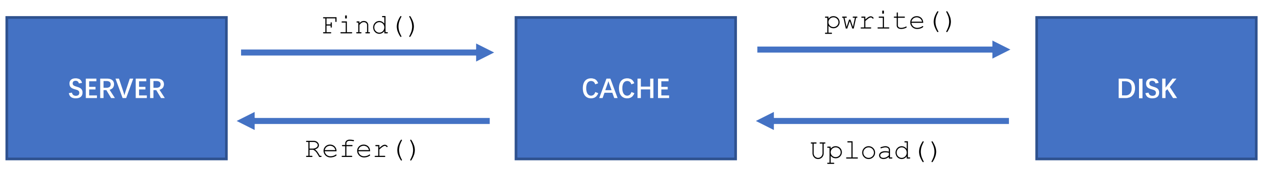
Functions	`Find(<httpname, 0>)`: check if the httpname in the cache_map `Refer(<httpname>,<cl>)`: read data from cache_map and send it out `Update(<httpname>)`: load file data from disk into cache block

GET	`if find(<httpname, 0>) != true then`
	`   update(<httpname>)`
	`end`
	`refer(<httpname>, <cl>)`

PUT	`update(<httpname>)`

For find function, the map provides build-in functions that can be used here.

The server and disk file should be linked by cache.



```
1.  Input    : char *argument <httpname> <cl>
2.  Shared List    : list cache, map cache_map
3.  list tem = new list
4.  for i to cache_size do
5.     auto iter = find(<std::pair(httpname, i>))
6.     tem.push_back(iter) // take this block out into a temporary cache block
7.     cache.erase(iter) // erase this block in cache
8.     Buffer = iter.value()
9.     send(cl, Buffer, sizeof(Buffer))
10. end
11. cache.splice(tem) // put the blocks used to the front of cache
```

Algorithm 4: refer() to read data from cache block

```
1.  Input    : char *argument <httpname>
2.  Shared List    : list cache, map cache_map
3.  for i to cache_size do
4.     auto iter = find(<std::pair(httpname, i>))
5.     Map.erase(<std::pair(httpname, i)>) // erase this key in cache_map
6.     cache.erase(iter) // erase this block in cache
7.  end
8.  fd = open(httpname)
9.  size_t index, offset
10. list tem = new list
11. while (read_result = pread(fd, buf, 0, offset)) != 0 do
12.    offset += read_result
13.    index = offset / 4KB
14.    tem.push_back(buf)  // store data into a temporary list
15.    Map.insert(<std::pair(httpname, index)>, *tem) // insert record into map with httpname
    and block_nr
16. end
17. // if the size is exceeded, then pop out the last recently used cache block with the same httpname
18. if (sizeof(tem) + sizeof(cache)) > cache_size do
19.    auto node = cache.pop_back() // erase cache block
20.    auto iter = map.find(<httpname, 0>) // erase key in map
21. end
22. cache.splice(tem) // put the blocks used to the front of cache
```

Algorithm 5: update() to load data from disk to cache

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

Update() makes sure four things: (1) the total block size should not exceed initialized block_size; (2) the correct data has been updated into the cache; (3) the data should be kept in the front of cache list; (4) the key should be inserted into map.

2.3 Synchronization and thread safety

As shown above, the multi-thread part has adopted mutex, semaphore, and conditional variables to make sure no threads can read/write when one thread is writing, and no thread can write when one thread is reading/writing. This keeps the shared variables was used sequentially but not concurrently.

For cache implementation, server must make sure all the threads must execute one by one but not concurrently. Since either the read from or write to cache will change the order of the list, the server must make sure the entire execution is executed under mutex that only one thread can access the cache at one time. Basically, add mutex controls in the beginning of and at the end of both refer() and update() to make sure the cache is synchronized.