Design Document: Multithreading and server caching

Yujia Li
CruzID: yli302

1 Goals

In assignment 1, I implemented an simple single-threaded HTTP client and server system, which implements PUT and GET features. In assignment 2, I use multi-threading to improve the throughput of server. To make server faster, an in-memory block cache is implemented.

2 Design

2.1 system design

The goal is to run a server like

```
./httpserver -N 8 -c 50 localhost:8888
```

2.1.1 multi-threading server

Default thread is 4, argument `-N nthreads` tells server create how many threads should be created.

A multi-threading server include a dispatch
In the main, dispatch thread will accept a descriptor from client, and pass this descriptor to a work thread if there is at least one not busy work thread, otherwise, wait. Dispatch thread do these things again and again. Work thread will wait if there has no job, and do work if dispatch thread hand off a descriptor to this work thread. Critical region is when change shared variables which tell threads to wait or not.(see algorithm part)

Another problem multi-threading server need to handle is

why your system is thread-safe. Which variables are shared between threads? When are they modified? Where are the critical regions?

2.1.2 in-memory block cache

Call cache.read ro cache.write

2.2 algorithms and data structures

Data structures

```
     /* use to store thread related information and share between
threads.
      * id is as same as the index of the array of thread_info
      * have_work use to tell thread work or wait.
      * cl is the file descriptor, and when it is -1, this work
thread is not busy.
      */
     struct thread_info{
          int32_t id;
          pthread_cond_t  have_work;
          int cl;
     }
```

cache : include a std::list, a preallocated struct block array, a std::map. Use to handle data, before read/write with disk. If cache contain expected data, read/write operation to disk will not happen.

```
     struct block{
         ssize_t sz;
         char data[4096];
         std::string httpname;
      };
```

Private variables:

```
  using key = std::pair<std::string, ssize_t>;
  using hashmap = std::map<key, block *>;

  hashmap cacheMap;
  std::list<block> cacheList;
  size_t cacheSize;
  block *blocks;
```

Functions of cache:

```
  // constructor
  LRUcache(size_t sz){
     cachesize = sz; blocks[sz];
     cachelist.assign(blocks, blocks+sz);
  }
  // if key exist return true.
  bool hasKey(httpname){
     if(httpname exist) return ture;
     else return false;
  }
```

```
// if cache is full return ture
bool isFull(){
    if(cacheList.size() > cacheSize) return true;
    else return false;
}
// set data to cache
void set(key, block){
    if(hasKey) evict all data with key->first; //httpname
    cacheList.push_front(&block);
    cacheMap.insert(pair<key, &block>);
    if(isFull) evict the blocks of file at back.
}
// get data from cache
void get(key, *buffer){
    if(hasKey)
         buffer = block[i].data;
}
```

Functions:

```
/* use getopt to parse argument (thread number, cache size)
 * -N for thread number, -c for cache size.
 */
getopt()
/* use pread and pwrite, don't need to consider data
consistency
 * and future sync when duel with server side files.
 * off += count
 */
pread(), pwrite()
```

Shared variables:

Shared variables	Act on
nworkers (semaphore)	To know whether at least a not busy work thread is exist. Initial working_thread as NUM_THREAD. Dispatch thread will wait if nworkers equal to zero.
have_work (condition variable)	This condition variable is in an array of struct

	thread_info[NUM_TRHEAD]. Dispatch thread will signal it after handing off descriptors.
lock_dispatch (mutex)	A lock related to dispatch work (descriptor) to work threads. Will lock nworkers and have_work.
nreaders(int32_t)	To know how many worker threads are operating read now.
is_writing (semaphore)	When write to local file, will wait this semaphore, when write finished, will signal this semaphore
lock_rw (mutex)	A lock related to read and write operation. Will lock nreaders, is_writing.

Shared data structure: cache.

operation	**Read**	**Write**
Read		**X**
Write	**X**	**X**

Algorithms and pseudocodes(include synchronize part):

pthread spawn and create thread

```
int main(){
      // (server socket setup)
      ……
      sem_init(nworkers, NUM_THREAD);
      pthread_mutex_init(lock_dispatch);
      nreaders = 0;
      sem_init(is_writing, 1);
      pthread_mutex_init(lock_rw);
      pthread_t thread[NUM_THREAD];
      thread_info tinfo[NUM_THREAD];
      for i = 0 to NUM_THREAD
            tinfo[i].id = i;
```

```
            pthread_cond_init(tinfo[i].have_work);
            tinfo[i].cl = -1;
            pthread_create(&thread[i],NULL,client_connection,
      &tinfo[i])
      end
      ……
      // (infinity loop of dispatch thread)
}
```

Dispatch thread

```
int main() {
      // (pthread spawn and create thread, socket setup)
      ……
      while(cl = accept())
            if (cl == -1) then
                  error; exit(1);
            End
            pthread_mutex_lock(&lock_dispatch)
            // sleep if all work threads are busy
            sem_wait(nworkers)
            pthread_mutex_unlock(&lock_dispatch)
            // check which work thread is not busy
            pthread_mutex_lock(&lock_dispatch)
            for i = 0 to NUM_THREAD
                  if (tinfo[i].cl == -1) then
                        tinfo[i].cl = cl;
                        pthread_cond_signal(&tinfo[i].have_work);
                  end
            end
            pthread_mutex_unlock(&lock_dispatch)
      end
}
```

work thread

```
void *client_connection(void *arg){
      struct thread_info *ti = arg;
      for (;;)
            pthread_mutex_lock(&lock_dispatch)
            //sleep if no work hand off from dispatch thread
            if (cl == -1) then
                  pthread_cond_wait(&ti.have_work, &lock_dispatch);
            end
            if (cl != -1) then
                  // nothing happen
            end
```

```
      pthread_mutex_lock(&lock_dispatch)

      /* handle HTTP header,
       * recv, send, read and write here.
       */
      // handle HTTP header
      ……
      /* PUT, only update the interaction with local file when
       * concurrency happen. In PUT, write has concurrency
problem
       */
      ……
      sem_wait(&is_writing);
      while(recv)
            cache.set(httpname, blocknum, buffer)
            write()
      end
      sem_post(&is_writing);
      ……
      /* GET, only update the interaction with local file when
       * concurrency happen. In GET, read has concurrency
problem
       */
      ……
      pthread_mutex_lock(&lock_rw);
      nreaders += 1;
      if(nreaders == 1) then
            sem_wait(&is_writing);
      end
      pthread_mutex_unlock(&lock_rw);
      // read
      if(cache.haskey(httpname)) then
            while(cache.find(httpname, blocknum, &buffer))
                  send(buffer);
                  blocknum += 1;
            end
      else
            while(pread())
                  send()
                  cache.set(httpname, blocknum, buffer);
            end
      end
      pthread_mutex_lock(&lock_rw);
```

```
            nreaders -= 1;
            if(nreaders == 0) then
                  sem_post(&is_writing);
            end
            pthread_mutex_unlock(&lock_rw);
            ……

            //set this worker thread as not busy
            pthread_mutex_lock(&lock_dispatch)
            sem_post(&nworkers);
            ti.cl = -1;
            pthread_mutex_unlock(&lock_dispatch)
      end
}
```