CSCI 6730 Operating System
Project 2 - Multi-threaded Web Server

Xiaodong Jiang
Department of Statistics
University of Georgia

1. Part I - Multi-threaded Web Server

   In this part, we are required to modify the single-threaded of web server to a multiple threaded version. The main idea is to modify the listener function in the original file. The Skeleton of the new listener function is as follows,

```
int listener(int port)
{
 /* keep the code in original version here*/
        while (1) {
                int s;
                s = accept(sock, NULL, NULL);
                if(getpeername(s, (struct sockaddr*) &peer, &peer_len) != -1) {
                printf("[pid %d, tid %d] Received a request from %s:%d\n", getpid(),
syscall(__NR_gettid) - getpid(), inet_ntoa(peer.sin_addr), (int)ntohs(peer.sin_port));
                        }
                pthread_t thread1;
                if (getpeername(s, (struct sockaddr*) &peer, &peer_len) != -1)
                {
                int r_thread1;
                r_thread1 = pthread_create(&thread1, NULL, (void *)&handler, s);
                if (r_thread1 != 0)
                {printf("%s\n", r_thread1);}
                }
        }
                close(sock);
}
```

   In the above code block, the blue marked code is to create new thread for each job accepted from the socket, which is very basic - first declare a new thread called thread1, then created with pthread_create function, however, instead of using the original process function, we define a new handler function (red marked above) as follows,

```
// self-defined handler
void handler(int s) {
        FILE *f;
        f = fdopen(s, "a+");
        process(f);
        fclose(f);}
```

Basically, we pass the parameter *s* to the handler function in *pthread_create()*, where *handler(int s)* will be executed in each thread. As long as one thread begins to work (Note, this job could finish very quick or last pretty long, which will not influence the next job), the *while(1)* block will continue listening the next signal from client end.

Thus, by modifying the listener function as well as creating a new handler function, we implement the multi-threaded web server as above. To test the performance, we first run our server with certain port, such as showing below with port 8888,

```
-bash-4.1$ ./webserver_part1 8888
HTTP server listening on port 8888
_
```

Then we test our server with htmlget_multi as given by Professor Lee, see the time cost with 10 and 100 threads, which is much faster than the single thread version.

```
-bash-4.1$ ./htmlget_multi localhost 8888 10
Time to handle 10 requests: 0.009591 sec
-bash-4.1$ ./htmlget_multi localhost 8888 100
Time to handle 100 requests: 0.070820 sec
```

2. Part II - Thread Pool Web Server

To address the the reason we implement thread pool for our web server, we first note that thread creation and termination are expensive operations that are repeated for every

request in Part I implementation, thus we will use thread pool to avoid such expensive operations.

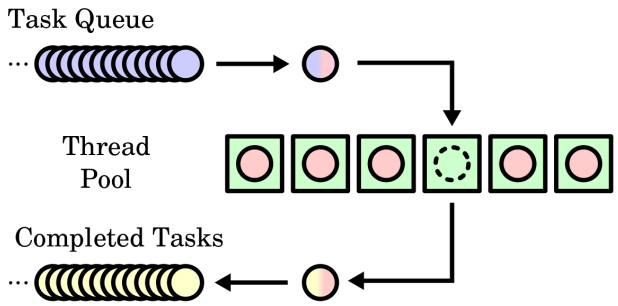
This part is relatively challenging than Part I, since we have to deal with (1). implement and maintain the thread pool including thread pool creation/initialization, add job, wait, and destroy, (2). data structure to maintain the jobs, such as linked list or queue, and (3). synchronization methods to avoid concurrency problems such as race conditions or deadlocks. We will introduce each of these points as follows.

First of all, let's first look at the overview of the functions and the thread pool's logical scheme as below,

Job Queue: | job1 | job2 | job3 | job4 | ...
ThreadPool: |Thead1 | Thread2 | Thread3| ...

Jobs are added to the job queue. Once a thread in the pool is idle, it is assigned with the first job from the queue(and it will be erased from the queue). It's each thread's job to

read from the queue serially(using lock) and executing each job until the queue is empty, which could also be visualized in the following diagram. (reference from wikipedia)



Implementation of thread pool

Now, let's first look at the important functions with detailed explanations to implement the thread pool as follows,

```
/*Initializes a threadpool. This function will not return untill all threads have initialized successfully.*/
struct thpool_* thpool_init(int num_threads)

/*Takes an action and its argument and adds it to the thread pool's job queue.*/
int thpool_add_work(threadpool, void (*function_p)(void*), void* arg_p);

/* it will wait for all jobs, including both queued and currently running to finish.
  As long as the queue is empty and all work has completed, the calling thread
  will continue.
*/
void thpool_wait(threadpool);

/* This will wait for the currently active threads to finish and then 'kill' the whole thread pool to free up memory.*/
void thpool_destroy(threadpool);
```

Binary semaphore as Synchronization

To avoid deadlocks or race condition, we implement binary semaphore as the synchronization method, where the internal counter of the semaphore can only take the values 1 or 0.

We first declare a semaphore and initialize it to the value of 1, After a semaphore is initialized, we can call one of two functions to interact with it, bsem_wait() or bsem_post(). The behavior of these two functions is described here:

```
/* Wait on semaphore until semaphore has value 0 */

static void bsem_wait(bsem* bsem_p) {
        pthread_mutex_lock(&bsem_p->mutex);
        while (bsem_p->v != 1) {
                pthread_cond_wait(&bsem_p->cond, &bsem_p->mutex);
        }
        bsem_p->v = 0;
        pthread_mutex_unlock(&bsem_p->mutex);}
```

```
/* Post to at least one thread.
increment the value of semaphore by 1
if there are 1 or more threads waiting, wake 1*/

static void bsem_post(bsem *bsem_p) {
        pthread_mutex_lock(&bsem_p->mutex);
        bsem_p->v = 1;
        pthread_cond_signal(&bsem_p->cond);
        pthread_mutex_unlock(&bsem_p->mutex);}
```

We then describe one important function thread_do to implement what each thread is doing, and avoid concurrency problems, and we only allow the execution of just ***one active thread*** from many others.

```
static void* thread_do(struct thread* thread_p){
        /*omit some code here */
        /* Mark thread as alive (initialized) */
        pthread_mutex_lock(&thpool_p->thcount_lock);
        thpool_p->num_threads_alive += 1;
        pthread_mutex_unlock(&thpool_p->thcount_lock);

        while(threads_keepalive){
                bsem_wait(thpool_p->jobqueue.has_jobs);
                if (threads_keepalive){
                        pthread_mutex_lock(&thpool_p->thcount_lock);
                        thpool_p->num_threads_working++;
                        pthread_mutex_unlock(&thpool_p->thcount_lock);

                        /* Read job from queue and execute it, omit code */

                        pthread_mutex_lock(&thpool_p->thcount_lock);
                        thpool_p->num_threads_working--;
                        if (!thpool_p->num_threads_working) {
```

```
                                pthread_cond_signal(&thpool_p->threads_all_idle);
                        }
                        pthread_mutex_unlock(&thpool_p->thcount_lock);
                }
        }
        pthread_mutex_lock(&thpool_p->thcount_lock);
        thpool_p->num_threads_alive --;
        pthread_mutex_unlock(&thpool_p->thcount_lock);
        return NULL;
}
```

Modification of the listener function (**Finally!!!**)

```
int listener(int port, int Num)
{
 /* keep the code in original version here*/
        // let's initialize the thread pool here
         threadpool thpool = thpool_init(Num);
         while (1) {
                int s;
                s = accept(sock, NULL, NULL);
                if(getpeername(s, (struct sockaddr*) &peer, &peer_len) != -1) {
                printf("[pid %d, tid %d] Received a request from %s:%d\n", getpid(),
syscall(__NR_gettid) - getpid(), inet_ntoa(peer.sin_addr), (int)ntohs(peer.sin_port));}

                if (getpeername(s, (struct sockaddr*) &peer, &peer_len) != -1)
                {thpool_add_work(thpool, (void*)handler, s);}
                }
                thpool_destroy(thpool);
                close(sock);
}

void handler(int s) {
        FILE *f;
        f = fdopen(s, "a+");
        process(f);
        fclose(f);
}
```

From above code block, we omit some code to save the space and highlight the important parts with blue color. We first initialize the thread pool with Num threads, where Num is given from command line argument (assume Num <= 100), then instead of creating new thread for each job, we simply add the job to thread pool by using

thpool_add_work function, where the handler function is still the same as Part I, to execute the job in each thread.
To test the performance, we use the similar way as Part I, we first run our server with certain port, such as showing below with port 7788, see below, creating 100 threads in thread pool,

```
-bash-4.1$ ./webserver_part2 7788 100
HTTP server listening on port 7788
THPOOL_DEBUG: Created thread 0 in pool
THPOOL_DEBUG: Created thread 1 in pool
THPOOL_DEBUG: Created thread 2 in pool
THPOOL_DEBUG: Created thread 3 in pool
THPOOL_DEBUG: Created thread 4 in pool
THPOOL_DEBUG: Created thread 5 in pool
THPOOL_DEBUG: Created thread 6 in pool
THPOOL_DEBUG: Created thread 7 in pool
THPOOL_DEBUG: Created thread 8 in pool
THPOOL_DEBUG: Created thread 9 in pool
THPOOL_DEBUG: Created thread 10 in pool
THPOOL_DEBUG: Created thread 11 in pool
THPOOL_DEBUG: Created thread 12 in pool
THPOOL_DEBUG: Created thread 13 in pool
THPOOL_DEBUG: Created thread 14 in pool
THPOOL_DEBUG: Created thread 15 in pool
THPOOL_DEBUG: Created thread 16 in pool
THPOOL_DEBUG: Created thread 17 in pool
THPOOL_DEBUG: Created thread 18 in pool
THPOOL_DEBUG: Created thread 19 in pool
THPOOL_DEBUG: Created thread 20 in pool
```

Then we test our server with htmlget_multi as given by Professor Lee, see the time cost with 10 and 100 threads, which is much faster than the single thread version.

```
-bash-4.1$ ./htmlget_multi localhost 7788 10
Time to handle 10 requests: 0.009928 sec
-bash-4.1$ ./htmlget_multi localhost 7788 100
Time to handle 100 requests: 0.071032 sec
```

3. Conclusion

In this project, we implement the multi-threaded web server and thread pool version, which is really challenging to me, especially the thread pool implementation, where lots of matters we should consider, such as special data structure, mutex lock, etc.

4. Reference

[1]. A minimal but powerful thread pool in ANSI C, github repository.
[2]. Class Notes from Professor REMZI H. ARPACI-DUSSEAU at UW Madison.
[3]. Textbook and Professor Lee's lecture notes.