EE488 Advanced programming techniques for Electrical Engineering

HW3:IMPLEMENTING TEXT SEARCH ENGINE

Files in text_search_engine folder:

1) client1.c
2) client2.c
3) common.h
4) helper.c
5) helper.h
6) Makefile
7) server.c
8) tc_malloc.c
9) tc_malloc.h
10) thread_pool.c
11) thread_pool.h

- helper.c,helper.h,tc_malloc.c,tc_malloc.h are files from previous assignments.
helper.c implements string functions and basic dynamic list and linked list implementation. In tc_malloc.c I have added tc_calloc(size_t and tc_realloc based on my tc_malloc and tc_free for the purpose of modularity.

The signature of the new functions are:

```
void * tc_calloc(size_t num,size_t size);

void * tc_realloc(void * ptr,size_t prev_size,size_t new_size);
```

- **I only used files from tc_malloc.c for creating packets as well as for any allocation purposes(to get the additional 60 pts).**

client1.c

- This client creates the requested amount of threads, and each thread requests the requested amount of requests to the server and they will receive the packet from server. I do not print anything for performance sake, unless the search word is not in the server's inverted index, which I will print not in the

server.
Implemented functions are:
- struct sockaddr_in *get_serveraddress(char *hostname, int port);
takes in hostname(which may be name or ip_address) and port and will return an allocated server address(struct sockaddr_in *).
- struct Hdr *create_pckt(size_t len, uint32_t msg_type);
takes in the length and message type and will return a packet with header.
- void *request_server(void *arg);
This is what each thread calls. This function takes in server_address and connects to server and requests a word then will receive a packet in return.

client2.c

- This client requests the server in an iterative fashion continuously and will print out the payload received from the server.
Implemented functions are:
(uses getserveraddress and create_pckt like client1) but request_server is a bit different as it requests the server continuously and each time we are supposed to type the word and the payload will be printed.

server.c

- This function implements an I/O multiplexing server with a thread pool for efficiency. It will wait for any request from a client and if it found one it will search from the inverted index which It made when it started, and returns a packet to a client. The packet may be a response or an error packet.
Implemented functions are:
- all of the functions from hw1 are used
- void thread_job(void * arg): This is the function that the each thread will call to do a job that is given from the listener thread. It takes in a file descriptor that is given from the listener thread, which the listener thread will use

select to get an event ,and will receive the request packet and send a response packet or an error packet if the search word is not in the inverted index.

common.h

- common.h file includes some functions that can be used by client and server. Such as a modified recv and write functions for efficiency and modularity and functions to manipulate a client or a server file descriptor. This file is shared by server.c, client1.c, and client.2.c.

    - function signature:
    int open_listenfd(int port);
    int open_clientfd(char *hostname, int port);
    int recv_from_host(int socket, char *buf, size_t len);
    int send_to_host(int socket, const char *buf, size_t len);

thread_pool.c and thread_pool.h

- I have implemented an easy queue data structure to store the jobs. The thread pool is implemented by functions:

static void * activate_workers();
This function is an infinite loop that the 10 threads are gonna execute. If there is no job then all threads are waiting for a signal from the listener thread and they are idle. But if there is some job then that job will be appended by the listener thread into a global work queue. Then one thread from all the 10 will access the mutex and dequeues from the queue. And will return the mutex after it fetches the job. Then that thread will call thread_job function to do the job.

int init_pool();
- This spawns 10 threads and they will call activate threads.

void add_job_to_pool(job_t *job);
- This function is called by the listener thread to enqueue a job. After the listener thread acquires the mutex lock it will enqueue the job and notify the workers in the activate_workers function.