System Programming MP4 report

B04902045 孫凡耘

December 11, 2016

Thread Pool Implementation

I initialized the thread pool when the server started by using pthread_create, number of threads created are specified in the config file. Every thread are created with a specified start routine(I named it handle_request_loop). I maintained a global queue, which stores all the request. I kept the IO multiplexing part from MP3 except that instead of handling the request immediately, the request is added to the global queue. Inside the function "handle_request_loop", there is a while loop that constantly detect whether the queeue is empty or not.If the queue isn't empty, handle the first element from the queue and pop the request from the queue. If the queue is empty, use pthread_cond_wait to wait for request to arrive. (I'll send a signal and awake one of the threads when a new request is added). There is a global varaible that counts how many threads are waiting for the signal. If the count is zero, then all threads are busy and the server returns "Server busy".

Processes and Threads

We can implement the server with multiple processes by using fork or vfork. For example, we can fork a new process when the server detects a new connection and let the new processes created be responsible for all requests from the specific client.

- Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process. Each thread has its own stack, and contex switching is faster.

- Threads in the same process share signals and signal handlers, while processes have to use IPC(Inter Process Communication) to communicate and work together.

- Since memory objects are shared among threads, shared data must be accessed through synchronization objects (like mutexes, condition variables and semaphores) that allow you to avoid data corruption.