Webserver with multi thread

Name: Zhihang Zhou
ID: zzhou37

Goal:

Upgrade the webserver to let it handle multiple connection at a time. The server can create N thread to handle multiple request and connection at the same time. If the thread doesn't have work to do, the main thread will put it to sleep and if there is something to do the main thread will wake it up. And during the process of handling request, the thread will put everything of what they did onto a logging file. If there is no thread are available. Then the main thread will put itself to sleep.

Design:

System design:

There is only one socket in the whole system, but the only one socket can accept multiple request and handling the request to a thread who is currently free. There is one main thread to open and control other service threads, and the other service thread can respond client request at the same time. When there is no free working thread, the working thread will put main thread to sleep.

Note: in here, each service thread is similar with the single thread client service model system.

For more detail description, the main thread will open N thread, then it will keep accepting different client requests and handle it to the service thread. If there is a thread have nothing to do, the semaphore will put it to sleep. If there is no client, the main thread will put itself to sleep. Beside handling requests, all the thread must write what they did in a log file. So, the log file should allow multiple thread to write it with the order of how they respond the client. But it doesn't have to respond with the order of client request.

The sharing data structure between different threads are three arrays, include an array of client, array of integer to represent if the thread is working or not and an array of semaphore can put each thread to sleep. Each working thread will stick with one box in each array. And the main thread will be looping through each box of the array at the same position each time and giving each working thread work to do.

For logging file, there is an int in the global variable and a semaphore control when a single thread can use it or not. So, when a thread needs to write on the log file, the thread must wait until there is no other thread using it at the same time. Then the thread will read the integer and int += how much space it needs to read or write into the file. Then, raise up the semaphore to let other working thread use it. Then the work thread will use the integer it got before it update as an off set to write Hex data into the file.

Inner connection of different functions:

Main function creates n threads and putting them to sleep, then create a semaphore array and an int array and a statue array to keep accepting request to feed them. The other function called single thread will accept requests, respond the request and write what it responds into a log file. Each of the single working thread will use the function from assignment one to respond one request to each client.

Structure of the system:

First, we need a shared data structure to share data between main thread and work thread. Second, we need a data structure can let the main thread know the statue of each working thread. Third, we need a structure can manage all the thread at the same time.

Hence, based on the previous requirement, I got the following solution:

N = 4				
Sem		logSem	int offset	
ID	0	1	2	3
Shared data structure:	cl	cl	cl	cl
Control structure:	sem(0)	sem(0)	sem(0)	sem(0)
Statue structure:	[0]	[0]	[0]	[0]
Thread:	T1	T2	T3	T4

In this example: everything is a global value and each thread are stick with a unique index, like the thread 0, can only read cl[0], call sem(0) down and change statue 0 to one. And the main function can modify everything here. But the main thread must work on one thread at a time. By the way, all the thread can call the logSem and the offset.

Here is something you might be interested, which is why I designed in this way. First the potential problem in synchronism operation system is dead log which is cause by more than one thread write data at the same time. Or one thread is reading data and other one is writing data. So, to prevent it happened and make design as simple as possible, I tried to let all thread independent to each other, so I only need to consider the interaction between main thread and each single thread at each time.

Detail:

Main thread: Main thread accept request and control all the thread:

As what I mentioned before, the main thread can accept a client and handle them to each working thread. So, there are following things must be prevented. Which is interrupt a thread which is working. Hence, what the main actually does is loop through the statue, if the statue of the thread is working then skip it. If the thread is not working, then accept a client and put it into the box of the corresponding index , then wake up the working the thread. Then call the main thread sem to go down.

The code:

Main
Allocate space for sem[]
Allocate space for statue[]
Allocate space for cl[]

Use for loop to initialize sem[]

Crate N thread and pass each argument into the corresponding working array

Initialize man sem to n -1

While true
If the statue[i] is 0
    cl[i] = accept request
    set statue[i] to 1
    wake up the sem[i]
    call down on mainstem
i++
i = i%n

the argument is a structure include an integer ID and a semaphore[ID] into argument.

Notice: in this case the main thread set statue[i] to 1 before waking up the I the thread, because it can prevent the loop enter the same thread at the second time.

Work thread: work thread responds

the work thread can accept each client request and respond the request of a client. But the client thread must be waked up by the main thread and must put itself to sleep after it finished the request. So, what it does is, call the semaphore and put it to sleep, when it was been waked up by the main function, then it will read the client from the corresponding box, then handling the request. After it finished the request, it will be modified it's statue to 0, and call up on the main semaphore to wake up the main thread.

Code:
Handling one request(argument is the package have everything we need in it)
//Open the package and copy the ID which is the index of thread to local
Copy int[ID]
Copy sem[ID]
Copy main[ID]
While
Sem[ID] wait
Respond the request of the client which is the old code from assignment one
Change the statue of it to 0
Call up the main semaphore

Log file: each working thread must be able to write logging information on the logging file

Log file include all the information of threads send to client, include the request name, file name and what it sends. But this assignment must allow all working threads can write on it al the same time without having overlapped information.
Therefore, in order to do that we must reserve space for each thread. In order to know how much space, we need, we need calculate how much space we need first.
First, find the size of header and the content length
Then find how many lines we need by using n / 20 char for each line + 1
Since each line transfer to Hex is 69 char, so the space we need will be line number * 69 + size of head and the end 7 =
After we know how much space we need, the next step will reserve space for it, we lock the offset integer first, then update the space for it. Then print head in the file. Next, when the thread sends or resave date form client, we read each line and write logging information into the file.

Code:
In the parse message function:

read content length
size = ((length / 20) + 1) * 69
size += size of head and end ===== line

after reserve space;
wait until the log file integer unlock by other semaphore
add the integer with the size we want to reserve

lock the log file lock
write the header on the file
if there is any bad request
    just write the error message on it
if not
//for put
read from client
call write log file function to write what we just read into log file
//for get
Write to client
call write log file function to write what we just read into log file

write log file function:
take argument buffer, seize need to be write, offset of the file, how much space in the last line as input and the line index number

if the last line is not full
    full up the last line

write each line into the log file
    first print the ith line number which is a 8-bit zero padded integer
    convert each line to 60 char long hex number

set the empty size of the last line to 69 – the size of last line