# Asgn2: Design Document

Alexander Soe cruzid: asoe

CSE130, Fall 2019

## Goal

The goal of this assignment is to add multithreading and logging to our httpserver. This means that if there are multiple clients, then our server should be able to simultaneously respond/handle multiple clients as well as log their requests . If a client PUT requests the server, the server should be able to create a file specified by the client and print the contents of the resource file from the client into the local file. On any other requests the server should respond with the appropriate status responses or error messages.

## Assumptions

I'm assuming that there is going to have to be some way to have the clients wait to be used by a thread if there are more clients than servers as well as there is going to have to be some way to make the threads wait for additional clients since none of the threads should have to be joined since the server should never stop running unless killed manually. As for logging there needs to be some sort of global offset and file descriptor as any thread should be able to write to the file at any point. To prevent deadlock, there has to be some sort of critical section that prevents any thread from taking a writing location from another. This is more efficient than having a lock on writing to the file since this implementation will make it possible for all threads to write to the file simultaneously.

## Design

Since we are using our previous assignment, all the necessary things to set up the http server using sockets will be predominantly the same. To implementing multithreading I'm going to create a queue or more specifically a c++ deque to hold my client file descriptors that I read from my socket side. The reason I'm using a deque instead of a vector is because I want the first client to connect to the server to be served first rather than the last one. A deque is more suitable to be used as a FIFO queue rather than a vector which is better at being a LIFO stack. Anyways as this queue is a shared resource between, I will need two sets of mutexes: one for the main thread to add clients to the queue and one for the other threads to access the queue for client file descriptors. Additionally there I will need a conditional variable for when the queue is empty. This is important as when clients get added to the queue, the condition variable is able to wake up a thread to respond to the client. To accomplish this task I'm going to make a context struct that contains a mutex, a condition variable and a deque. I'm going to create an instance of the struct and pass it as the parameter cast as a void ptr to all my create pthread calls and then re-cast it as a context struct so that the deque is shared between the main function and the worker threads.

As for the logging, I'm going to create a global log file descriptor as well as a global offset. Any time a thread tries to the log file they must obtain an offset that they can write in. To prevent threads from writing over each other there needs to be a mutex to stop other threads from

obtaining the same writing space. As GET logs and Error logs will always be roughly the same length, they will be pretty easy to log as their sizes will be easy to calculate. On the other hand logging PUT messages is going to be a little trickier as you have to find how many lines of 20 each PUT log will create based on the commands content length. Additionally the line numbers must be brought into the size as well as the \n and the ending "=" signs that will terminate the log. But after finding this algorithm printing is pretty simple as all you have to do is convert each character into a hex number using sprintf and printing each individual character to the logfile. You also have to watch out where you end off after each log file print, because you have to make sure that all the lines are length 20 and you have to remember to some extent where you are left after each write.
As for errors, I will use sscanf to check if the argument given to -N is a number or else I will return an error if the thread argument is not a number. Additionally, if a -l option is given then, another error will be thrown for the lack of a log file name. I'll also have a flag to check if a log file exists to make sure that I don't try to print to one if it doesn't. All the sections that could cause problems because of shared resources are protected with mutexes and a condition variable. As with the last assignment if the command is not a PUT or GET or the connection for some reason cuts while the server is trying to handle a put, then a 500 status error message will be sent. If a file can't be accessed, then a 403 error will be issued and if a file just straight up doesn't exist and a GET request is issued then 404 will be issued. All of these error messages will be written into the log file as failures if one exists.

## Pseudocode

```
Struct context
{
        Pthread_cond_t cond;
        Pthread_mutex_t lock;
        Queue clientQueue;
}

//Create a lock to stop other threads from stealing each others space in the logfile
Size_t getOffSet(size_t logLength)
{
        lock(mutex);
        currLogPos  = offset;
        offSet +=  logLength;
        unlock(mutex);
        Return currLogPos;
}
```

```
Size_t printLogHeader(char fileName, size_t contLength)
{
	//Calculate the total offset using the ceiling function
	//Write "========\n" into the the last 9 spots in the offset for the logfile
	//Write the PUT header message
}

String convertCharToHex(char c)
{
	hex[3];
	sprintf(hex,  " %02X", c);
	String hexString = hex;
	Return hexString;
}

String convertLineNumber()
{
	//Same as convertCharToHex but with the lineNumbers
}

Size_t printPUTLog(size_t totalReadSize, size_t readSize, char fileContents, size_t startPosition)
{
	Size_t newPosition = startPosition;
	for(int i = 0…..)
	{
		if(totalReadSize % 20 == 0)
		{
			pwrite(convertLinenumber);
			startPosition += convertLineNumber.length();
		}
		pwrite(convertCharToHex);
		startPosition += converCharToHex.length();
		if(totalReadSize % 20 == 19)
		{
			pwrite("\n");
			startPosition++;
		}
	}
	Return newPosition;
}

Void printGetLog(char fileName)
{
	//print the getLog header
}
```

```
Void pringErrorLog(char fileName)
{
	//print error Log
}

String readHeader(int fd)
{
	while(read(fd) > 0)
	{
		if(strstr(“\r\n\r\n”) != nullptr)//check for end of header
		{
			break;
		}
	}
	if(n < 0)
	{
		close(fd);
		Return string
	}
	Return string
}

Int getContentLength(string header)
{
	strstr(“Content-Length:”) //find content length keyword
	if(Content-length is there)
	{
		sscanf(“Content-Length: %d”)
		If(contentLength > 0)
		{
			Return length;
		}
	}
	Else
	{
		Return error
	}
	Return -1;
}
```

```
Bool isCorrectFileName(string filename)
{
	if (filename is greater than or less than 27)
	{
		Return false;
	}
	if(characters are the right ascii values)
	{
		Return false;
	}
	Return true;
}

String getFileName(string header)
{
	Vector tokenVector;
	Stringstream tokStream;
	String token;
	while(getline(tokstream))
	{
		tokenVector.push_back(token);
	}
	String filename = tokenVector[1];
	Return filename;
}

Void handlPut(filename, contentLength, client_fd)
{
	if(access(file access))
	{
		if (access(write access))
		{
			Write 403 response
		}
	}
	if(fileExists and there is contentlength)
	{
		Write 200 response
	}
	if(!fileExits and contentLength)
	{
		Write 201 response
```

```
        }


        if(contLength != -1)
        {
                create(file);
                while(contLength > 0)
                {
                        read(client);
                        printPutLog(...);
                        write(filecontents to newfile);
                }
        }
        close(newfd);
        Else
        {
                read(client);
                printPutLog(...);
                write(till eof)
        }
}

Void handleGet(filename, client_fd)
{
        printGETLog(...);
        if (access(file exists))
        {
                Write 404;
        }
        if (access(read ok))
        {
                Write 403;
        }
        write(client_fd, header length);
        while(contentlength != 0)
        {
                Read(fileContents);
                write(client_fd);
        }
}

handle()
{
        while(there is a header)
        {
```

```
                //Call get
                //Call put
                //Check for errors
        }
}

Int main()
{
        //Do all the socket setup stuff
        //Call all the file
        while(1)
        {
                while(clientfd = accept >0)
                {
                        lock(mutex);
                        push(client fd);
                        Cond_signal;
                        unlock(mutex);
                }
        }
        Return 0;
}
```