Group: Habanero
Daniel Engel, Alex Lindsay, William Massey, Robert Knight

Client:

For the client side we check the command line arguments for the IP and port address as well as for the scriptname. If no script name is present it will enter interactive mode which it will function like a command prompt. If a script name is present it will read that file. For both routes the user input is then cleaned by removing excess whitespace at the start, end, and in the middle. It will also convert any tabs to spaces as necessary. It will then execute these commands. If no IP and Port are given it will reference the .clientrc for that information, which by default is 127.0.0.1:3000. The client can perform three different commands: get, list, and quit. Quit will terminate the end of the client. List will output all the files in the directory that was chosen when running the server. The get command will get the media files from the server.

Server:

For the server you run it with arguments for the port number, number of threads, buffer length, scheduling algorithm, and the directory. We check to make sure that all of these arguments are present before executing anything further with the server. Since for this project we only have one scheduler you can only use FIFO.

server:

Server defines main. In addition, there is a function that takes the ip address and port number and converts them into a string for the user. After this, they are stored in a global char *ipp. This is used by the header class to make it easy and convenient to send the ip address and port number when the header is sent to the client. Beyond this, a global abstract Scheduler *sched is defined. When the server is initializing itself, it calls a factory that determines the kind of scheduler passed in and then returns the particular scheduler. sched points to that particular scheduler. This allows the server to interface with the scheduler without needing to know all of the nuances of that particular scheduler.

dispatch:

The dispatcher is run as its own thread. It uses a semaphore in order to manage the total number of threads running at a time. When a thread becomes available, the dispatcher calls the scheduler method NextTask, which then returns the next TaskInfo object. That is then run as a thread.

scheduler:

There are two kinds of schedulers. This first is an abstract scheduler that defines basic methods. The second is a set of particular schedulers that inherit from the abstract scheduler. This allows us to easily and conveniently work with different kinds of schedulers while also not needing to modify the server in order to handle the particularities of this or that kind of scheduler. The servers only interaction with the scheduler is to simply add tasks. Actually running the tasks is the purview of the dispatcher, which runs as a separate thread.

scheduler_FIFO:

In this particular assignment, only FIFO scheduling needs to be implemented. FIFO stands for “First In First Out”, and indicates that tasks are completed in whatever order they are entered. There is no sense of prioritizing some tasks over other tasks. The queue is by definition a FIFO data structure. Because of that, the Scheduler_FIFO class has a std::queue data member that is used to hold the set of tasks. Anytime the dispatcher asks for a task, the Scheduler_FIFO pops the queue, and then returns the TaskInfo *.

scheduler_factory:

In C++, one can use a pointer to a base class to point to the various derived classes. That allows one to create a layer of abstraction over the various scheduler types. In doing this, the server is able to work with a variety of schedulers without needing to know anything about the schedulers that are being used. The scheduler factory takes a string that represents the type of scheduler that was provided over the command line. It uses a string compare and then based on the results returns the appropriate scheduler. If there is no match for any of the available schedulers, then nullptr is returned. This allows the scheduler to return with error if the command line option is not a valid scheduler type.

task:

The TaskInfo class contains all of the information necessary to run a certain task. It has a pointer to its thread, which allows it to alert the scheduler that its thread is finished. It also has the file descriptor of the client that it is to write to. The other two member variables are its priority and a pointer to a Header object.

The priority member variable is not relevant for the FIFO scheduler. Instead, it can be used for the SJF and SRJF schedulers. The lower the value of the priority, the higher priority it is. A priority of 0 is given to any task that is assigned an invalid command. The reason for this is because an invalid command only sends back a header. It does not have a payload associated. This means that an invalid command will generally be the shortest job.

A priority of 1 is given to any task that is assigned a list command. A list command will send back a string that contains the contents of the media directory. Unless there are thousands of files in the media directory, a list command will usually be shorter than any get command. In the future, the priority can be changed so that it is equal to the number of characters to write.

Any get commands will have a priority that is equal to the size of the file that is requested. If the file size is 1KiB, then the priority will be 1024. Bigger files will be longer jobs, and therefore will be selected after smaller files. A good data structure that could make it very convenient to get the shortest jobs for the SJR and SRJF schedulers is the std::multimap. The difference between the std::multimap and the std::map is that the multimap allows for duplicate keys.

When a new task is to be run, DoRequest is called. DoRequest is provided a TaskInfo pointer. DoRequest takes the associated task, and hands the file descriptor and the Header pointer to WriteRequest which actually performs the request. Once WriteRequest returns, DoRequest deletes the TaskInfo object.

WriteRequest calls a method provided by the Header that consolidates all of the header information into a single string. After that, WriteRequest checks to see if the command is valid. If the command is not valid, then it simply writes the header to the file descriptor and cleans up. If the command is valid, then it checks to see if the command is list or get. If the command is list, then writes the header and then contents of gti.listing. If the command is a get command, then it opens the file provided by get and writes that file into the file descriptor. After that it cleans up.

task.h defines a struct referred to as a global_task_info struct. An instance of this struct is then created globally. The global_task_info struct contains information that is generally important for all tasks to have. The two primary members are the directory path and a listing of the files in that directory

header:

The Header class contains all of the information that is provided in the header that is initially sent to the client. The first member is the response integer, which indicates whether or not the request was valid. The second is the ip address and port number of the server. The third is the type of request (binary versus text). The fourth is the length of the request, which is used in order to establish priority. The fifth is the request itself (the name of the file).

Regarding the request itself, it is only useful with get commands currently. This is because the directory listing of the server is considered to be static. Were files to be deleted or added to this server, then request would probably contain a char * for the directory listing (assuming a list command was provided). Currently, it just holds the name of the file that the client wants to get. This should be changed in the future so that it accounts for files that are added or deleted to the server.

One notable method in the Header class is GetTotalString. This takes all of the header information and turns it into a single string that can then be written to the client. This returns a std::string in order to take advantage of the convenience that std::string provides with concatenation.

sem:

sem.h defines two semaphores: one for the number of accepted connections (a_sem), and then one for the number of threads (t_sem). These are initialized in initSemaphores. sem_wait(a_sem) is called in server.cpp, before accept. sem_post is called after sem_wait(t_sem). sem_post(t_sem) is called during the return of DoRequest.

GetFile:

GetFile defines a single function list, which returns a string containing the the contents of the directory.

Appendix A:

Makefile

```
CC  = g++
LDFLAGS = -lm -lnsl -lpthread
CFLAGS  = -ggdb
TARGET  = server client

default: $(TARGET)

server: server.o
    g++ $(CFLAGS) -o server server.cpp sem.cpp FileGet.cpp commandLineCheck.cpp
dispatch.cpp scheduler_factory.cpp task.cpp scheduler.cpp scheduler_FIFO.cpp
header.cpp $(LDFLAGS)

client: client.o
    g++ $(CFLAGS) -o client client.cpp commandLineCheck.cpp lineParser.cpp
request.cpp getLine.cpp excessParser.cpp $(LDFLAGS)

clean:
    -rm -f *.o *~

cleanall: clean
    -rm -f $(TARGET)
```

Client.cpp

```
#include <string>
#include <vector>
#include <cstring>
#include "getLine.h"
#include "lineParser.h"
#include "request.h"

char *ip_addr = NULL;
```

```
in_port_t port;

void doInteractive() {
    ///interactive
    std::string interactiveCommand;
    std::cerr << ">>";
    while (std::getline(std::cin, interactiveCommand)) {

        if (interactiveCommand.compare("quit") == 0)
            break;

        lineParser(interactiveCommand);
    std::cout << ">>";
    }
}

bool checkDefaultIPP(int sd) {
    // Check clientrc file in order to determine if
    // ip address and port are valid
    std::ifstream client(".clientrc", std::ifstream::in);

    if (!client.is_open()) {
        return false;
    }

    std::string ip; std::string p;

    getline(client, ip, ':');
    getline(client, p);

    struct sockaddr_in server;
    memset((char *) &server, 0, sizeof(struct sockaddr_in));
    server.sin_family = AF_INET;
    port = (in_port_t) atoi(p.c_str());
    server.sin_port = htons(port);
    int result = inet_aton(ip.c_str(), &server.sin_addr);

    if (!result) {
        return false;
    }

    if (connect(sd, (struct sockaddr *) &server,
            sizeof(server)) < 0) {
        return false;
    }

    ip_addr = strdup(ip.c_str());
    port = server.sin_port;

    close(sd);
    return true;
}

bool checkIPP(std::string ip, std::string p) {
    // Check if the provided port and ip address is valid
    struct sockaddr_in server;
    memset((char *) &server, 0, sizeof(struct sockaddr_in));
    server.sin_family = AF_INET;
    port = (in_port_t) atoi(p.c_str());
    server.sin_port = htons(port);
    int result = inet_aton(ip.c_str(), &server.sin_addr);

    int sd = socket(AF_INET, SOCK_STREAM, 0);

    if (sd == -1) {
        return false;
    }

    if (!result) {
        return checkDefaultIPP(sd);
    }

    if (connect(sd, (struct sockaddr *) &server,
            sizeof(server)) < 0) {
        return checkDefaultIPP(sd);
    }

    ip_addr = strdup(ip.c_str());
    port = server.sin_port;

    close(sd);
    return true;
}


int main(int argc, char *argv[]) {

    if (argc == 2){
        doInteractive();
    } else {
        getLine(argv[2]);
    }

    return 0;
}
```

commandLineCheck.cpp

```
#include "commandLineCheck.h"

bool directoryCheck(char *path)
{
    DIR* dir = opendir(path);
    if (dir)
    {
        closedir(dir);
    return true;
    }
    return false;

}
```

commandLineCheck.h

```
#ifndef COMMAND_LINE_CHECK_H
#define COMMAND_LINE_CHECK_H
```

```
#include <dirent.h>
#include <errno.h>
#include <string.h>

bool directoryCheck(char *path);

#endif
```

Dispatch.cpp

```
#include "dispatch.h"

void *dispatch(void *d) {
    TaskInfo *curr_task = nullptr;
    pthread_t thread;

    // Wait until a task finishes and then move to next
    while (1) {
        if (sched->Task_Count() > 0) {
            // Decrement the semaphore
            sem_wait(t_sem);
            // Run the new task
            curr_task = sched->Next_Task();
            // Now that we are running thread, allow new connection
            sem_post(a_sem);
            pthread_create(&thread,
                    NULL,
                    DoRequest,
                    (void *) curr_task);
        }

    }

    return nullptr;
}
```

Dispatch.h

```
#ifndef DISPATCH_H
#define DISPATCH_H
#include <pthread.h>
/* The dispatcher is what actually runs the scheduler */
#include "scheduler.h"
#include "sem.h"

extern sem_t *t_sem;
extern sem_t *a_sem;

extern Scheduler *sched;

/* The dispatcher takes the number of threads as a parameter.
 * Pass the address of threads and type cast to void *
 *
 * Nothing is returned
 */
void *dispatch(void *);

#endif
```

excessParser.cpp

```
#include "excessParser.h"

excessParser::excessParser() : result("") { }

excessParser::~excessParser() { }

void excessParser::doStrip(std::string line) {
    std::string temp;

    // Parse string and set result equal to final output
    temp = stripComments(line);
    temp = tabToSpace(temp);
    temp = stripOuter(temp);
    temp = stripInner(temp);
```

```cpp
    result = temp;
}

std::string excessParser::getParsedString() {
    return result;
}

std::string excessParser::stripComments(std::string line) {
    /* Determine whether or not there are any comments to strip.
     * This means getting the index of the first # found. However,
     * it must be determined that # is not part of a quote or string
     * literal.
     */

    bool found_comment = false;
    bool in_quote = false;
    bool in_literal = false;
    size_t i;

    for (i = 0; i < line.size(); ++i) {
        // If we have found ", but didn't find ' first
        if (line[i] == '\"' && !in_literal) {
            // Reverse whether we are in quote or not.
            // This allows us to track opening vs. closing "
            in_quote = !in_quote;
        }

        // Same for here.
        else if (line[i] == '\'' && !in_quote) {
            in_literal = !in_literal;
        }

        // We found a comment and are not inside a quote. Exit
        else if (line[i] == '#' && !in_literal && !in_quote) {
            found_comment = true;
            break;
        }
    }
    // Return the original line if we never found a comment
    if (!found_comment) {
        return line;
    }

    std::string stripped;

    // strip the comment
    stripped = line.substr(0,i);

    return stripped;
}

std::string excessParser::tabToSpace(std::string line) {
    // Turn all tabs into spaces
    for (size_t i = 0; i < line.size(); ++i) {
        if (line[i] == '\t')
            line.replace(i, 1, 1, ' ');
    }

    return line;

}

std::string excessParser::stripOuter(std::string line) {
    // Remove leading whitespace

    while (line[0] == ' ') {
        line.erase(0,1);
    }

    // Remove trailing whitespace
    while (line[line.size() - 1] == ' ') {
        line.erase(line.size() - 1, 1);
    }

    return line;
}
```

```cpp
std::string excessParser::stripInner(std::string line) {
    bool found_whitespace = false;

    for (size_t i = 0; i < line.size(); ++i) {
        /* If we found the first whitespace between words,
         * then set the bool. We'll ignore the first one
         */
        if (!found_whitespace && line[i] == ' ')
            found_whitespace = true;

        /* If we had already found the first whitespace between
         * words, then we'll erase this whitespace.
         */
        else if (found_whitespace && line[i] == ' ')
            line.erase(i, 1);

        /* If we find a non whitespace character, then we are
         * inside of a word. Go ahead and set found_whitespace to
         * false. Once we exit the word, we'll go back to stripping
         * whitespace.
         */
        else if (line[i] != ' ')
            found_whitespace = false;
    }

    return line;
}
```

excessParser.h

```cpp
#ifndef EXCESS_PARSER_H
#define EXCESS_PARSER_H
#include <sstream>
#include <string>

/* Parse and remove excess white space and comments */
class excessParser {
```

```
public:
    // Do nothing
    excessParser();
    // Do nothing
    ~excessParser();
    void doStrip(std::string line);
    std::string getParsedString();
private:
    // Strip comments
    std::string stripComments(std::string);
    // Convert each tab to a single space
    std::string tabToSpace(std::string);
    // Strip leading and trailing whitespace
    std::string stripOuter(std::string);
    // Strip whitespace between words
    std::string stripInner(std::string);

    std::string result;
};

#endif
```

FileGet.cpp

```
#include "FileGet.h"

std::string list() {
    DIR *dir = opendir(gti.path);
    struct dirent* d;
    std::string files = "";
    if (dir)
    {
        while ((d = readdir(dir)) != NULL)
        {
            files = files + d->d_name;
            files.append(" ");
        }
        closedir(dir);
    }
    else
    {
        /*Directionary does not exist*/
        fprintf(stderr, "list: %s\n", strerror(errno));
    }

    files[files.length()-1] = '\n';
    return files;
}
```

FileGet.h

```
#ifndef FILE_GET_H
#define FILE_GET_H
#include <dirent.h>
#include <errno.h>
#include <string>
#include <cstring>
#include <cstdio>
#include <sys/types.h>
#include "task.h"

extern struct global_task_info gti;

std::string list();

#endif
```

getLine.cpp

```
#include "getLine.h"

void getLine(std::string commandFile)
{
    std::ifstream inFile;
    std::string commands;
    inFile.open(commandFile);
    while(getline(inFile, commands))
    {
        if(commandCheck(commands))
        {
            lineParser(commands);
        }
    }
    inFile.close();
}

bool commandCheck(std::string commands)
{
    for(size_t i = 0; i < commands.length(); i++)
    {
        if(!std::isspace(commands[i]))
        {
            if(commands[i] == '#')
            {
                return false;
            }

            return true;
        }
    }
    return false;
}
```

getLine.h

```
#ifndef GET_LINE_H
#define GET_LINE_H
#include "lineParser.h"
#include <fstream>
#include <sstream>
// Daniel: This is being added to check shebang
#include <unistd.h>
#include <algorithm>
```

```
void getLine(std::string commandFile );
bool commandCheck(std::string commands);


#endif
```

Header.cpp

```
#include "server.h"
#include "header.h"
#include "task.h"

Header::Header(int resp, const char *ipp,
            const char *type, size_t length,
            const char *request)
    : resp(resp), length(length) {

    // Get the size of ipp and type
    size_t ipp_size = strlen(ipp);
    size_t type_size = strlen(type);
    size_t req_size = strlen(request);

    // Allocate on the heap
    this->ipp = new char [ipp_size+1];
    this->type = new char [type_size+1];
    this->request = new char[req_size+1];
    // Add null terminators to the end
    this->ipp[ipp_size] = 0;
    this->type[ipp_size] = 0;
    this->request[req_size] = 0;

    // Copy
    this->ipp = strcpy(this->ipp, ipp);
    this->type = strcpy(this->type, type);
    this->request = strcpy(this->request, request);
}

Header::~Header() {
    delete [] ipp; ipp = nullptr;
    delete [] type; type = nullptr;
    delete [] request; request = nullptr;
}

int Header::GetResponse() {
    return this->resp;
}

char *Header::GetRequest() {
    return this->request;
}

/* Create a formatted string out the header.
 * This will be given to WriteRequest
 */
std::string Header::GetTotalString() {
    // Convert response and length to strings
    char response_buff[4] = {0,0,0,0};
    char *response_ptr = &response_buff[0];
    int length = snprintf(NULL, 0, "%lu", this->length);
    char length_buff[length+1];
    char *length_ptr = &length_buff[0];
    // Clear the array
    memset(length_ptr, 0, length+1);
    snprintf(response_ptr, 3, "%d", this->resp);
    snprintf(length_ptr, length, "%lu", this->length);

    std::string retval(response_ptr);
    retval.append("\n");
    retval.append(ipp);
    retval.append("\n");
    retval.append(type);
    retval.append("\n");
    retval.append(length_ptr);
    retval.append("\n");
    return retval;
}

char *Header::GetType() {
    return this->type;
}

size_t Header::GetLength() {
    return this->length;
}

Header *MakeHeader(const char *req) {
    char *request = strdup(request);
    // Set up header
    Header *h = nullptr;
    // Fill header if list
    if (strncmp(request, "list", 4) == 0) {
        h = new Header(100,
                ipp,
                "text",
                gti.listing.size()+1,
                "list");

        free(request);
        // Return the header
        return h;
    }

    /* Fill header if get
     * The request field will have the name of the request
     * and will not include "get"
     */
    else if (strncmp(request, "get", 3) == 0) {
        // Get the size of request
        size_t request_size = strlen(request);
        struct stat st;
        // Remove trailing whitespace
        for (size_t i = request_size - 1; i != 0; --i) {
            if (request[i] == '\t' ||
                request[i] == ' ' ||
                request[i] == '\n' ||
                request[i] == '\0') {
                request[i] = 0;
            }

            else {
                break;
            }
        }

        // Convert tabs to spaces
        for (size_t i = 0; i < request_size; ++i) {
            if (request[i] == '\t') {
                request[i] = ' ';
            }
        }

        // Get a pointer to the last space
        char *what = strrchr(request, ' ');
        // Point to character after last space
        ++what;
        // Determine the size of the file
        stat(what, &st);

        // Make sure that the file exists
        if (errno != ENOENT) {
            // Fill the necessary information
            h = new Header(100,
                    ipp,
                    "binary",
                    st.st_size,
                    what);

            free(request);
            // Return the new header
            return h;
        }
    }

    // If we got to this point, then that means that something went wrong.
    h = new Header(-100, ipp, "invalid", 0, "invalid");
```

```
    free(request);
    return h;
}
```

Header.h

```
#ifndef HEADER_H
#define HEADER_H
#include <cstdio>
#include <cstring>
#include <string>
#include <sys/stat.h>
#include <errno.h>
#include "task.h"

class Header {
public:
    Header(int resp, const char *ipp, const char
            *type, size_t length, const char *request);
    ~Header();
    int GetResponse();
    std::string GetTotalString();
    char *GetType();
    char *GetRequest();
    size_t GetLength();
private:
    int resp;
    char *ipp;
    char *type;
    size_t length;
    char *request;
};


/* When the task is created, it calls make header.
 * Make header determines what kind of request is being made.
 * MakeTask will use GetResponse in order to determine what
 * kind of task is being performed
 */
Header *MakeHeader(const char *request);

#endif
```

lineParser.cpp

```
#include <string>
#include <iostream>
#include <vector>
#include <sstream>
#include "lineParser.h"
#include "excessParser.h"
#include "request.h"

void lineParser(std::string lineInput){
    excessParser ep;
    std::stringstream stream;

    if (lineInput.empty()) {
        return;
    }

    ep.doStrip(lineInput);
    lineInput = ep.getParsedString();

   int fd = sendRequest(lineInput);

   if (lineInput.find("get") != std::string::npos) {
    stream << lineInput;
    // Get the file that is to be extracted
    std::getline(stream, lineInput, ' ');
    receiveRequest(fd, lineInput);

    return;
   }
   receiveRequest(fd, lineInput);


}
```

lineParser.h

```
#include <string>
#include <iostream>
#include <vector>
#include <sstream>
#include "request.h"


void lineParser(std::string inputLine);

#endif
```

Request.cpp

```
#include "request.h"

int sendRequest(std::string req) {
    /* Connect to server */
    struct sockaddr_in server;
    int sd = socket(AF_INET, SOCK_STREAM, 0);

    if (sd == -1) {
        return -1;
    }

    memset((char *) &server, 0, sizeof(struct sockaddr_in));
    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    int result = inet_aton(ip_addr, &server.sin_addr);

    // Somethign went wrong
    if (!result)
        return -1;
```

```cpp
    if (connect(sd, (struct sockaddr *) &server,
            sizeof(server)) < 0) {
        return -1;
    }

    write(sd, req.c_str(), strlen(req.c_str()));

    // Return the file descriptor
    return sd;
}

void receiveRequest(int sockfd, std::string filename) {
    size_t bufferlen = 256;
    size_t filelength = 0;
    char buffer[bufferlen];
    char *b_ptr = &buffer[0];
    std::stringstream stream;
    std::string temp;
    bool is_text = false;
    bool after_header = false;
    memset(b_ptr, 0, bufferlen);
    std::ofstream output;

    // Read the header first
    int n = read(sockfd, b_ptr, bufferlen);
    stream << b_ptr;
    std::getline(stream, temp, '\n');
    int header_value = 0;
    std::stringstream value(temp);
    value >> header_value;

    if (header_value == 100) {
        after_header = true;
    }

    std::cout << temp << std::endl;
    std::getline(stream, temp, '\n');
    if (temp.find("text") != std::string::npos) {
        is_text == true;
    }

    std::cout << temp << std::endl;

    std::getline(stream, temp, '\n');

    value.clear(); value.str("");

    value << temp;

    value >> filelength;

    std::cout << temp << std::endl;

    // Prepare to write any remaining info before going back to reading
    int header_size = stream.tellg();
    b_ptr = buffer + header_size;

    if (!is_text) {
        output.open(filename, std::ofstream::binary);
        output << b_ptr;
    }

    else {
        std::cout << b_ptr;
    }

    b_ptr = &buffer[0];

    memset(b_ptr, 0, bufferlen);

    while ((n = read(sockfd, b_ptr, bufferlen)) > 0) {
        if (is_text) {
            std::cout << b_ptr;
        }

        else {
            output << b_ptr;
        }

        memset(b_ptr, 0, bufferlen);
    }

    if (is_text) {
        std::cout << std::endl;
    }

    else {
        output.close();
    }

    close(sockfd);
}
```

Request.h

```
#ifndef REQUEST_H
#define REQUEST_H
#include <string>
#include <sstream>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fstream>

extern char *ip_addr;
extern in_port_t port;

int sendRequest(std::string req);

void receiveRequest(int sockfd, std::string filename);
```

```
#endif
```

Scheduler_factory.cpp

```
#include "scheduler_factory.h"

Scheduler *MakeScheduler(const char *scheduler) {
    /* If any, default to FIFO */
    if (strncmp(scheduler, "ANY", 3) == 0) {
        return new Scheduler_FIFO();
    }

    /* If FIFO, return FIFO scheduler */
    if (strncmp(scheduler, "FIFO", 4) == 0) {
        return new Scheduler_FIFO();
    }

    /* We have not implemented other schedulers */
    else {
        return new Scheduler_FIFO();
    }

}
```

Scheduler_factory.h

```
#ifndef SCHEDULER_FACTORY_H
#define SCHEDULER_FACTORY_H
#include <cstring>
#include "scheduler_FIFO.h"

extern Scheduler *sched;

Scheduler *MakeScheduler(const char *scheduler);

#endif
```

scheduler_FIFO.cpp

```
#include "scheduler_FIFO.h"
#include "server.h"

Scheduler_FIFO::Scheduler_FIFO() : Scheduler() { }

/* Empty the queue if there is anything remaining */
Scheduler_FIFO::~Scheduler_FIFO() {
    TaskInfo *t = nullptr;
    while (!this->q.empty()) {
        t = this->q.front();
        this->q.pop();
        delete t;
    }
}

size_t Scheduler_FIFO::Task_Count() {
    return this->q.size();
}

TaskInfo *Scheduler_FIFO::Next_Task() {

    TaskInfo *t = this->q.front();
    this->q.pop();
    return t;
}

void Scheduler_FIFO::Add_Task(TaskInfo *t) {
    this->q.push(t);
}
```

scheduler_FIFO.h

```
#ifndef SCHEDULER_FIFO_H
#define SCHEDULER_FIFO_H
#include <errno.h>
#include <cstdio>
#include <pthread.h>
#include <queue>
```

```cpp
#include "scheduler.h"
#include "task.h"

/* Scheduler FIFO inherits from schedule */
class Scheduler_FIFO : public Scheduler {
public:
    Scheduler_FIFO();
    ~Scheduler_FIFO();
    virtual TaskInfo *Next_Task();
    virtual void Add_Task(TaskInfo *);
    virtual size_t Task_Count();
/* Since we're doing FIFO, we will use a queue */
private:
    std::queue<TaskInfo *> q;
};

#endif
```

Scheduler.cpp

```cpp
#include "scheduler.h"

Scheduler::Scheduler() { }

Scheduler::~Scheduler() { }
```

Scheduler.h

```cpp
#ifndef SCHEDULER_H
#define SCHEDULER_H
#include <cstddef>
#include <vector>
#include <pthread.h>
#include "task.h"

/* The server will work with an abstract scheduler */
class Scheduler {
public:
    Scheduler();
    virtual ~Scheduler();
    virtual TaskInfo *Next_Task() = 0;
    virtual void Add_Task(TaskInfo *) = 0;
    virtual size_t Task_Count() = 0;
};

#endif
```

Sem.cpp

```
#include "sem.h"

const char *sem_accepted = "/accepted";
const char *sem_threads = "/threads";
sem_t *a_sem = nullptr;
sem_t *t_sem = nullptr;


bool initSemaphores(unsigned int threads, unsigned int buffers) {
    if (buffers > threads)
        a_sem = sem_open(sem_accepted, O_CREAT, 660, buffers - threads);
    else
        a_sem = sem_open(sem_accepted, O_CREAT, 600, buffers);

    if (a_sem == SEM_FAILED) {
        fprintf(stderr, "%s\n", strerror(errno));
        return false;
    }

    t_sem = sem_open(sem_threads, O_CREAT, 660, threads);

    if (t_sem == SEM_FAILED) {
        fprintf(stderr, "%s\n", strerror(errno));
        sem_close(a_sem);
        return false;
    }

    return true;
}
```

Sem.h

```
#ifndef SEM_H
#define SEM_H
#include <pthread.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <string.h>
#include <cstdio>
#include <errno.h>

// Names of the semaphores
extern sem_t *a_sem;
extern sem_t *t_sem;

bool initSemaphores(unsigned int threads,
            unsigned int buffer);

#endif
```

Server.cpp

```
#include "server.h"

Scheduler *sched = nullptr;

char *ipp = NULL;

void SetIPAddr(struct in_addr, in_port_t);

int main(int argc, char **argv)
{
    int     n, bytes_to_read;
    int     sd, new_sd;
    unsigned int    client_len;
    int port;
    unsigned int threads;
    unsigned int num_bufs;
    struct  sockaddr_in     server, client;
```

```
char    *bp, buf[BUFLEN];
pthread_t thread;
bp = &buf[0];
// Clear the buffer
memset(bp, 0, BUFLEN);

// Set arguments

if (argc != 6) {
    fprintf(stderr, "Usage: ./server [portnum] [threads]");
    fprintf(stderr, " [buffers] [sched] [directory]\n");
    exit(1);
}

// Convert port, thread, and buffers to integers
port = (int) strtol(argv[1], NULL, 10);
threads = (unsigned int) strtol(argv[2], NULL, 10);
num_bufs = (unsigned int) strtoul(argv[3], NULL, 10);

if (!initSemaphores(threads, num_bufs)) {
    fprintf(stderr, "Failed to set semphores!\n");
    exit(1);
}

// Use the scheduler factory to set sched
sched = MakeScheduler(argv[4]);

if (!sched) {
    fprintf(stderr, "Invalid scheduler\n");
    exit(1);
}

// Check if media directory is valid
if (!directoryCheck(argv[5])) {
    fprintf(stderr, "Invalid directory\n");
    delete sched;
    exit(1);
}
```

```
    // Set global task info struct's path member
    gti.path = argv[5];
    gti.listing = list();

    /* Run the dispatcher for the scheduler in a separate thread */
    pthread_create(&thread, NULL, dispatch, (void *) &threads);

    /* Create a stream socket */
    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket\n");
        exit(1);
    }

    /* Bind an address to the socket */
    bzero((char *)&server, sizeof(struct sockaddr_in));
    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    server.sin_addr.s_addr = htonl(INADDR_ANY);

    /* Set the ascii form of the ip address and port.
     * This will be used by the header class
     */
    SetIPAddr(server.sin_addr, server.sin_port);


    if (bind(sd, (struct sockaddr *)&server,
    sizeof(server)) == -1) {
        fprintf(stderr, "Can't bind name to socket\n");
        exit(1);
    }

    /* queue up to 5 connect requests */
    listen(sd, 5);

    while (1) {
        // Get semaphore for next connection
        sem_wait(a_sem);
        client_len = sizeof(client);
        if ((new_sd = accept(sd, (struct sockaddr *)&client,
        &client_len)) == -1) {
            fprintf(stderr, "Can't accept client\n");
            exit(1);
        }

        bp = buf;
        bytes_to_read = BUFLEN;
        while ((n = read(new_sd, bp, bytes_to_read)) > 0) {
            bp += n;
            bytes_to_read -= n;
        }

        bp = &buf[0];
        memset(bp, 0, BUFLEN);
        // Copy into a string for convenience
        std::string req(bp);

        // Make new task
        TaskInfo *t = MakeTask(new_sd, req.c_str());
        // Add task into the queue and let scheduler
        // handle running it
        sched->Add_Task(t);
    }

    delete sched; sched = nullptr;
    close(sd);
    return(0);
}

void SetIPAddr(struct in_addr addr, in_port_t port) {
    // Set the address
    char *address = inet_ntoa(addr);
    // Get the size
    size_t addr_size = strlen(address);
    // Get the length of the port number
    size_t port_len = snprintf(NULL, 0, "%u", port);
    // Create a buffer and 0 the buffer
    char port_buff[port_len+1];
    char *pb_ptr = &port_buff[0];
    memset(pb_ptr, 0, port_len+1);

    // Copy the port number
    snprintf(pb_ptr, port_len, "%u", port);

    /* Allocate enough memory for the ip address,
     * the port number, and also a color
     */
    ipp = new char[addr_size + port_len + 2];

    // Copy the ip address into ipp
    ipp = strcpy(ipp, address);
    // Add the colon
    ipp[addr_size] = ':';
    // Copy the port into ipp now
    char *ipp_ptr_2 = strcpy(&ipp[addr_size+1], pb_ptr);
}
```

Server.h

```
#ifndef SERVER_H
#define SERVER_H
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include "scheduler.h"
#include "dispatch.h"
#include "commandLineCheck.h"
#include "scheduler_factory.h"
#include "sem.h"
#include "FileGet.h"
```

```
extern sem_t *t_sem;
extern sem_t *a_sem;

#define SERVER_TCP_PORT 3000
#define BUFLEN      256

/* ip address of server, as well as its port */
extern char *ipp;

void SetIPAddr(struct in_addr, in_port_t);

#endif
```

Task.cpp

```
#include "task.h"
#include "server.h"
#include "sem.h"

struct global_task_info gti;

TaskInfo::TaskInfo(int sockfd, const char *request)
    : sock_fd(sockfd) {

    this->h = MakeHeader(request);

    /* Since bad requests only involve sending a header back,
     * We're going to give it the highest priority.
     * The reason for this is that sending a bad request
     * header back to the client will take shorter than
     * sending a request header along with some file
     * or listing of the media directory
     */
    if (this->h->GetResponse() != 100)
        this->priority = 0;

    /* If the reqeuest is a listing, this gets a priority of 1
     * The reason for this is that it will generally be quicker
     * to send the directory listing than a binary file
     */
    else if (strcmp(this->h->GetType(), "list") == 0)
        this->priority = 1;

    /* The only other command is get. For that reason, the
     * priority is based on the size
     */

    else
        this->priority = this->h->GetLength();
}

TaskInfo::~TaskInfo() {
    delete h; h = nullptr;
    close(this->sock_fd);
}

Header *TaskInfo::GetHeader() {
    return h;
}

size_t TaskInfo::GetPriority() {
    return this->priority;
}

int TaskInfo::GetSockFD() {
    return this->sock_fd;
}

void TaskInfo::SetThread(pthread_t *thread) {
    this->thread = thread;
}

TaskInfo *MakeTask(int fd, const char *request) {
    TaskInfo *t = new TaskInfo(fd, request);
    return t;
}
```

```
void *DoRequest(void * t) {
    // Get the header and the file descriptor
    TaskInfo *task = (TaskInfo *) t;
    Header *h = task->GetHeader();
    int fd = task->GetSockFD();

    // Cal writerequest
    WriteRequest(fd, h);
    // Delete the task now that we're finished
    delete task;
    // Release semaphore
    sem_post(t_sem);
    return nullptr;
}

void WriteRequest(int fd, Header *h) {
    /* Go ahead and initialize the kinds of strings
     * that may be used in the function
     */
    int char_read = 0;
    std::string header = h->GetTotalString();
    std::string request = h->GetRequest();
    std::string path(gti.path);
    char buffer[256];
    char *b_ptr = &buffer[0];
    memset(b_ptr, 0, 256);
    /* Add the / to the end of the path if needed */
    if (path.back() != '/')
        path.append("/");
    // Get the header contents and write it out
    write(fd, header.c_str(), header.size()+1);

    /* Exit if the request was not valid */
    if (h->GetResponse() != 100) {
        return;
    }

    /* Print list of directories if the request is list */
    if (strncmp(request.c_str(), "list", strlen(request.c_str()))) {
        write(fd, gti.listing.c_str(), gti.listing.size()+1);
    }


    // Open the file that is to be sent over the network
    int readfd = open(path.c_str(), O_RDONLY);

    if (readfd < 0) {
        fprintf(stderr, "Failed to open %s: %s\n",
                path.c_str(), strerror(errno));
        close(fd);
    }

    /* Continually read from the file that was just opened
     * and write it to the socket.
     *
     * When there is eof of error, the transfer will stop.
     */
    while ((char_read = read(readfd, b_ptr, 256)) > 0) {
        write(fd, b_ptr, char_read);
    }

    /* Don't worry about closing the socket file descriptor.
     * That is taken care of in TaskInfo's destructor.
     */
    close(readfd);
}
```

Task.h

```
#ifndef TASK_H
#define TASK_H
#include <dirent.h>
#include <pthread.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string>
#include <cstdio>
#include <errno.h>
```

```
#include <cstring>
#include "header.h"
#include "sem.h"

extern sem_t *t_sem;
extern sem_t *a_sem;

/* This is information that is shared by all tasks.
 * This includes directory information */
struct global_task_info {
    /* This is the path to the media directory.
     * The reason it is a char * instead of a string
     * is that having it as a char * allows it to be
     * set directly from argv
     */
    char *path;
    /* A listing of contents in the media directory
     * The professor did not mention the media directory
     * ever changing. Hence, we can simplify things by
     * setting the listing at the beginning.
     */
    std::string listing;
};

class TaskInfo {
public:
    // The constructor will set priority and header
    TaskInfo(int sock_fd, const char *request);
    ~TaskInfo();
    Header *GetHeader();
    size_t GetPriority();
    int GetSockFD();
    void SetThread(pthread_t *thread);
private:
    int sock_fd;
    Header *h;
    size_t priority;
    pthread_t *thread;
};
```

```
// The server will set gti
extern struct global_task_info gti;

TaskInfo *MakeTask(int fd, const char *request);

/* DoRequest takes the provided task info object as a parameter.
 * It calls write request, providing the file descriptor and the header.
 *
 * Once the request is complete, DoRequest deletes the task info object
 */
void *DoRequest(void *);

/* write request writes the header to the fd, and then it writes either
 * the file itself or the listing (or nothing if it's a bad request
 */
void WriteRequest(int fd, Header *h);

#endif
```