

CS 140 Project 2

Documentation

A.Y. 2021-2022 Sem 2

Hans Gabriel H. De Castro

2019-03184

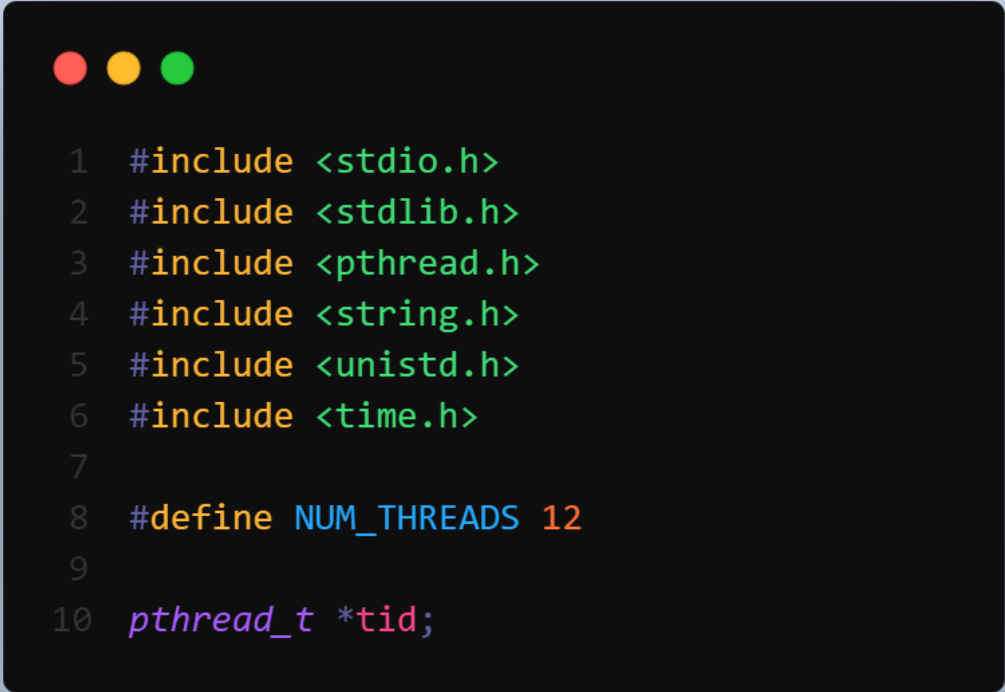
CS 140 GITING 7

Declaration

The File server that was implemented only reached level 3.

Implementation

The implementation has been divided into 10 sections to make the discussion more organized. These 10 sections are seen in the 10 figures below. Note that the line numberings do not match the actual numbering in the program's c file. Note that the source code is file_server.c



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <string.h>
5  #include <unistd.h>
6  #include <time.h>
7
8  #define NUM_THREADS 12
9
10 pthread_t *tid;
```

Figure 1 Included libraries and global vars



```
1  typedef struct arguments
2  {
3      char command[50];
4      char path[50];
5      char string[50];
6  }ARGS;
```

Figure 2 arguments struct



```
1  int get_random()  
2  {  
3      // Returns a pseudo-random integer  
4      // range is [0,100)  
5  
6      int r = rand()%100;  
7      return r;  
8  }
```

Figure 3 get random function

```
1 void random_sleep(int op)
2 {
3     int random = get_random();
4     if(op == 1) // sleep for 1 or 6 seconds
5     {
6         if(random < 20)
7         {
8             sleep(6);
9         }
10        else
11        {
12            sleep(1);
13        }
14    }
15    else if(op == 0) // random delay between 7 to 10 seconds
16    {
17        if(random < 25)
18        {
19            sleep(7);
20        }
21        else if(random >= 25 && random < 50)
22        {
23            sleep(8);
24        }
25        else if(random >= 50 && random < 75)
26        {
27            sleep(9);
28        }
29        else if(random >= 75)
30        {
31            sleep(10);
32        }
33    }
34 }
35 }
```

Figure 4 random sleep function

```

1 void readfile(ARGS *arguments)
2 {
3     FILE *f = fopen(arguments->path,"r");
4     //Error exit if there was a problem opening the above file
5     if (f == NULL) {
6         fprintf(stderr, "Error opening file %s \n", arguments->path);
7         return;
8     }
9
10    //Seek to the end of the file and error exit if fseek() indicates an error condition
11    if (fseek(f, 0, SEEK_END) == -1) {
12        fprintf(stderr, "Error seeking in file %s", arguments->path);
13        return;
14    }
15
16    //Get the position of the end of the file
17    int file_size = ftell(f);
18    //Reset the file position back to the beginning of the file
19    rewind(f);
20    //Allocate a character buffer the size of the file plus one more byte for the null terminator
21    char *buf = malloc(file_size + 1);
22
23    //Error exit if malloc couldn't allocate memory for the buffer
24    if (buf == NULL) {
25        fprintf(stderr, "Error allocating memory for file %s \n", arguments->path);
26        return;
27    }
28
29    if (fread((void *) buf, file_size, 1, f) <= 0) {
30        if(strcmp(arguments->command,"read") == 0)
31        {
32            FILE *output = fopen("read.txt","a");
33            fprintf(output,"%s %s:\tFILE DNE\n",arguments->command,arguments->path);
34            fclose(output);
35        }
36        else if(strcmp(arguments->command,"empty") == 0)
37        {
38            FILE *output = fopen("empty.txt","a");
39            fprintf(output,"%s %s:\tFILE ALREADY EMPTY\n",arguments->command,arguments->path);
40            fclose(output);
41        }
42        free(buf);
43        return;
44    }
45
46    //Set the null terminator at the end of the buffer
47    buf[file_size] = '\0';
48    if(strcmp(arguments->command,"read") == 0)
49    {
50        FILE *output = fopen("read.txt","a");
51        fprintf(output,"%s %s:\t%s\n",arguments->command,arguments->path,buf);
52        fclose(output);
53    }
54    else if(strcmp(arguments->command,"empty") == 0)
55    {
56        FILE *output = fopen("empty.txt","a");
57        fprintf(output,"%s %s:\t%s\n",arguments->command,arguments->path,buf);
58        fclose(output);
59    }
60
61    //Cleanup
62    free(buf);
63    fclose(f);
64 }

```

Figure 5 readfile function

```

1 void worker_write(ARGS *arguments)
2 {
3     ARGS *args = (ARGS *)arguments;
4     random_sleep(1);
5     FILE *fp_write = fopen(args->path,"a");
6     fprintf(fp_write,"%s",args->string);
7     usleep((25*strlen(args->string))*1000); // multiply to 1000 to get 25 ms
8     fclose(fp_write);
9     free(args);
10    pthread_exit(NULL);
11 }

```

Figure 6 worker_write function

```

1 void worker_read(void *arguments)
2 {
3     ARGS *args = (ARGS *)arguments;
4     random_sleep(1);
5     if(access(args->path,F_OK) == 0)
6     {
7         readfile(args);
8     }
9     else
10    {
11        FILE *output = fopen("read.txt","a");
12        fprintf(output,"%s %s:\tFILE DNE\n",args->command,args->path);
13        fclose(output);
14    }
15
16    free(args);
17    pthread_exit(NULL);
18 }

```

Figure 7 worker_read function

```

1 void worker_empty(void *arguments)
2 {
3     ARGV *args = (ARGV *)arguments;
4     random_sleep(1);
5     if(access(args->path,F_OK) == 0)
6     {
7         readfile(args);
8         FILE *fp = fopen(args->path,"w");
9         fclose(fp);
10        random_sleep(0);
11    }
12    else
13    {
14        FILE *output = fopen("empty.txt","a");
15        fprintf(output,"%s %s:\tFILE ALREADY EMPTY\n",args->command,args->path);
16        fclose(output);
17    }
18
19    free(args);
20    pthread_exit(NULL);
21 }

```

Figure 8 worker_empty function

```

1 void save_command(char *buf)
2 {
3     time_t T = time(NULL);
4     struct tm tm = *localtime(&T);
5     FILE *commands = fopen("commands.txt","a");
6     fprintf(commands,"%02d/%02d/%04d %02d:%02d:%02d %s\n",tm.tm_mday, tm.tm_mon + 1, tm.tm_year
+ 1900,tm.tm_hour, tm.tm_min, tm.tm_sec,buf);
7     fclose(commands);
8 }

```

Figure 9 save_command function


```

1  int main()
2  {
3      srand(time(0));
4      tid = malloc(sizeof(pthread_t)*NUM_THREADS);
5      int thread_counter = 0;
6      void *fs[] = {worker_write, worker_read, worker_empty};
7      char buf[150];
8      printf("$ ");
9      scanf("%[^\n]s", buf);
10     save_command(buf);
11     while(1)
12     {
13         if(strcmp(buf, "exit") == 0)
14         {
15             break;
16         }
17         else
18         {
19             char cmd[50];
20             char path[50];
21             char string[50];
22             sscanf(buf, "%s %s %s", cmd, path, string);
23             if(strcmp(cmd, "write") == 0)
24             {
25                 ARGV *args = (ARGV *)malloc(sizeof(ARGV));
26                 strcpy(args->command, cmd);
27                 strcpy(args->path, path);
28                 strcpy(args->string, string);
29                 pthread_create(&tid[thread_counter], NULL, fs[0], (void *) args);
30                 thread_counter++;
31             }
32             else if(strcmp(cmd, "read") == 0)
33             {
34                 ARGV *args = (ARGV *)malloc(sizeof(ARGV));
35                 strcpy(args->command, cmd);
36                 strcpy(args->path, path);
37                 pthread_create(&tid[thread_counter], NULL, fs[1], (void *) args);
38                 thread_counter++;
39             }
40             else if(strcmp(cmd, "empty") == 0)
41             {
42                 ARGV *args = (ARGV *)malloc(sizeof(ARGV));
43                 strcpy(args->command, cmd);
44                 strcpy(args->path, path);
45                 pthread_create(&tid[thread_counter], NULL, fs[2], (void *) args);
46                 thread_counter++;
47             }
48             else if(strcmp(cmd, "thread_count") == 0)
49             {
50                 printf("thread_counter = %d\n", thread_counter);
51             }
52             else
53             {
54                 printf("wrong command!\n");
55             }
56             printf("$ ");
57             scanf("%[^\n]s", buf);
58             save_command(buf);
59         }
60     }
61     free(tid);
62     return 0;
63 }

```

Figure 10 main function

Discussion of the Implementation

The key idea in this implementation is to use the thread running in the main function as the master thread and from there, the worker threads will be spawned based on the input. The main function is shown in figure 10. Line 3 (line 193 in file) calls `srand` for the generating a random number later. Line 4 (line 194 in file) in figure 10 dynamically allocates space for an array of `pthread` thread ids. This array of threads is declared in line 10 (and in file too) of figure 1. Dynamic allocation was used because the number of worker threads to be spawned is unknown. The size initially allocated is 12 multiplied to `NUM_THREADS`, which can be seen in figure 1 (line 8 in file). `NUM_THREADS` is set to 12 because my CPU has 12 threads. To aid this, we see in line 5 (line 195 of file) of figure 10 that `thread_count` is initialized to 0. This tells the current index that should be accessed in `tid` to spawn a new thread. This can be seen later in `pthread_create`. In line 6 (line 196 in file) of figure 10, we see an array of the void functions that worker_threads will be sent to. In line 7 (197 in file) of figure 10, we see the declaration of the char array named `buf`. This will catch the inputs in the terminal later. The reason why it has the size of 50 is because it is expected that the maximum length of the file path and the string input are 50. The remaining space is for the command, but an overhead was given just to be sure. line 8 (line 198 in file) just prints “\$ ” to serve as a prompt. Line 9 (line 199 in file) takes an input from the terminal and stores that as a string in `buf`. What’s weird in this `scanf` is that it will only stop reading characters until it sees the newline character. In this way, the input does not halt when it sees a space. Hence, it is possible to store the string “write file.txt hello” or “read file.txt” or “empty file.txt”. Line 10 (line 200 in file) calls the function named `save_command`. This function is responsible for logging or storing the read input into `commands.txt` attached with a timestamp. This function is defined in figure 9. In figure 9, we can see in lines 3-4 (lines 184-185 of file) that the local time of the system is stored in `tm`. In line 5 (line 186 of file) that `commands.txt` is being opened with permission set to append. In line 6 (line 187 of file), the timestamp and the entire command stored in `buf` is appended to `commands.txt`. And then the file pointer is closed. Going back to main in Figure 10, we can see in line 11 (line 201 in file) that there is a while loop that loops indefinitely. Inside this loop, in line 13 (203 in file) is an if statement that checks whether the `buf` is equal to “exit”. If the input is “exit”, then we break out of the while loop. Otherwise, we proceed to line 19 (209 in file). Lines 19-21 (209-211) will store the command (write, read or empty), file path, and the string that will be written into the three char arrays named `cmd`, `path`, and `string` respectively. Line 22 (212 in file) reads `buf` and splits `buf` into several strings separated by a space. With this, the command, file path, and string are stored into the previously mentioned char arrays. Line 23 (213 in file) checks whether the command is “write”. If it is, we proceed to line 25 (215 in file). This file dynamically allocates space for struct `ARGS` `args`. `ARGS` is a struct defined in figure 2 (line 12 in file). In figure 2, we can see in lines 3-5 (14-16 in file) that the struct has 3 fields. These are command, path, and string. They just store the relevant information for the command. In figure 10 line 26-28 (216-218 in file), we see that the inputs are being stored into the relevant fields of `args`. In line 29 (219 in file), a new thread is created

using `pthread_create`. We can see here that the index in `tid` being accessed is determined by `thread_counter`. We can also see that the index in `fs` being accessed is 0 because this refers to `worker_write` which is what we need. And lastly, we passed `args` as an argument to `worker_write` when we spawn this new worker thread. We then increment the `thread_counter` in line 30 (220 in file). If the command is not write, then in line 32 (222 in file), we check whether the command is “read”. If it is, then we proceed to line 34 (224 in file). Lines 34-36 (224-226 in file) are similar to that of 25-28 (215-218) of write. But `strcpy(args->string,string)` is missing because there is no string input for the read command. Only the read command and the path of the file to read from are needed. The only difference between line 37 (227 in file) and 29 (219 in file) is that the function that the new thread will be spawned into is `fs[1]` which refers to `worker_read`. And then we increment `thread_counter` again. If the command is not “write” and “read”, then in line 40 (230 in file) we check if it is “empty”. If it is then we proceed. Lines 42-46 (232-236) are the similar to that of read but the difference is in line 45 (235 in file). The index being referred to in `fs` is now 2 because this refers to `worker_empty`. If the command is not write, read, and empty, we check in line 48 (238 in file) if it is equal to “`thread_count`”. This command is used to check how many threads were spawned so far by printing the value of `thread_counter`. If the commands do not match the previously mentioned cases then we print “wrong command!”. After all these, we then repeat lines 8-10 (198-200) and loop again to accept input. Lastly, line 62 (252 in file) just frees up `tid`.

We saw earlier that a worker thread is spawned to the function `worker_write` when the command is “write”. We now look at the `worker_write` function in figure 6 (line 129 in file). In line 3 (131 in file) of the said function, the arguments parameter was just renamed to `args` for ease of typing. Line 4 (132 in file) calls the function `random_sleep` with input 1. This function is defined in figure 4 (line 28 in file) and what it does when the passed input is 1, is it either sleeps for 1s (80% probability) or 6s (20% probability). Its implementation will be discussed later. Line 5 (133 in file) opens the file specified in the file path with permission set to append. And then in line 6 (134 in file), the input string is appended into the recently opened file. Line 6 (135 in file) adds a delay of 25 ms per character. The reason why the input of `usleep` is `25*strlen(args->string)*1000` is because the input of `usleep` is actually in microseconds. And 1 milliseconds = 1000 microseconds. Line 8 (136 in file) just closes the file pointer and line 9 (137 in file) frees the dynamically allocated space for the arguments. And lastly, line 10 (138 in file) forcefully terminates the worker thread.

We also saw earlier that a worker thread is spawned to the function `worker_read` when the command is “write”. We now look at the `worker_read` function in figure 7 (line 141 in file). Similar to `worker_write`, it also has arguments and it is renamed to `args`. there is also `random_sleep(1)` with the same purpose. In line 5 (145 in file), we check determines whether a file path can be accessed (0 if yes, otherwise no). If it can be accessed, we proceed to line 7 (147 in file) to call the function `readfile` and pass in `args`. This function is defined in figure 5 (line 64 in file) and what it basically does for

worker_read is it reads the file specified by the args->path and it writes the contents of the file to read.txt. However, the function also checks whether the file is empty and it writes to read.txt the appropriate response. The full implementation of readfile will be discussed later. Going back to worker_read, lines 11-13 (151-153 in file) only runs when the file cannot be accessed. Line 11 (151 in file) just opens the read.txt file with permission set to append. Line 12 (152 in file) appends to read.txt the entire command and then "FILE DNE" which means that the file cannot be accessed. And then lines 16-17 (156-157 in file) are just for cleaning up and termination of the thread.

We also saw earlier that a worker thread is spawned to the function worker_empty when the command is "empty". We now look at the worker_empty function in figure 8 (line 160 in file). It is almost like worker_read. The major differences are lines 8-10 (167-169 in file) and 14-15 (173-174 in file). We can see in line 7 (166 in file) that readfile was also called but the difference here with worker_read calling readfile is that the contents of the args->path will be written to empty.txt instead of read.txt. After that, the file specified by args->path will be opened with permission set to write so that the file will be emptied. And then it is closed. Line 10 (169 in file) calls random_sleep but with input 0. What this does is instead of sleeping for either 1s or 6s with 80% vs 20% probability, the thread will sleep between 7 to 10 seconds instead. Line 14 (173 in file) opens the file empty.txt instead of read.txt and in line 15 (174 in file), the entire command and "FILE ALREADY EMPTY" is appended to empty.txt.

The function readfile is extremely important for the implementation of the read and empty command. Its implementation can be seen in figure 5 (line 63 in file). Lines 67-70 just checks whether there is an error when opening the file specified in args/arguments->path and it writes an error to stderr. The reason why it writes to stderr instead of just writing into empty.txt is because args->path was already checked in either worker_read or worker_empty if the file can be accessed or not. If it still fails in readfile, then something must be really messed up and it should alarm the tester. Lines 11-13 (lines 74-77) performs fseek to seek the location of the end of the file and prints and writes to stderr when it encounters a problem. Line 17 (80 in file) obtains the size of the file. And line 19 (82 in file) resets the position of the file pointer. Line 21 (84 in file) dynamically allocates space to store the contents of the file to be read to buf. The size is file_size + 1 because we will save an extra space for the null terminator. Lines 24-26 (87-89) just checks whether malloc failed and writes the appropriate message to stderr and exits. Line 29 (92 in file) stores all the contents of the file into buf (it includes even whitespaces and newline characters). And then if the return value of fread is less than or equal to 0, it means that the file is empty, or an error happened and the following lines just write to either read.txt or empty.txt depending on the issued command if it is read or empty the appropriate message and then we exit. Line 47 (110 in file) just appends a null terminator in buf. Lines 48-58 (111-121) writes the contents of arguments->path (stored in buf) into the appropriate destination. It writes to read.txt if the issued command is read and it writes it to empty.txt if the issued command is empty. The remaining lines are just for cleaning up.

The function `random_sleep` was mentioned multiple times earlier. Its implementation can be seen in figure 4 (line 28 in file). We can see that it has an input named `op`. its value is either 1 or 0 and we will see later how it affects the behaviour of the function. Line 3 (30 in file) gets a pseudo-random number in the range [0,100) using the function `get_random` whose implementation can be seen in figure 3 (line 19 in file). Note that `get_random` was provided in `uvle`. Line 4 of figure 4 checks if the input `op` is equal to 1. If it is then it means that we will either put the calling thread to sleep for 6 seconds if the obtained random number is less than 20 for 20% probability or make the thread sleep for 1 second if it is greater than or equal to 20 for 80% probability. If `op` is not 0 then in line 15 (42 in file), we check if it is equal to 0. If it is, then we want to make the calling thread sleep for either 7,8,9, or 10 seconds with all having equal 25% probability. That is why in lines 17, 21, 25, and 29 (44, 48, 52, and 56), we split the range [0,100) into 4 sections.

The struct `args` was already discussed earlier. We can see its implementation in figure 2 (line 12 in file). It is simple. It has 3 fields: (1) command, (2) path, and (3) string. These fields will store the parts of an input given by the user. We know that the format of input when the command is `write` is like this: `write <file path> <string input>`. We store `write` into command, file path to path, and string input to string.

Lastly, Figure 1 just shows all the included libraries that were necessary for the implementation of this project and `NUM_THREADS` and `tid`.

Testing

Note that in all of the tests that will be done, we will assume that `command.txt`, `read.txt`, and `empty.txt` all exists but are empty and are located inside the directory where `file_server` is (`/home/hans/CS_140/Proj2`). We can see this in figure 11 below.

The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar displays a project structure for 'Proj2 (WSL: Ubuntu)'. The files listed include `test.txt`, `a.txt`, `commands.txt`, `empty.txt`, `file_server`, `file_server.c`, `proj2`, `proj2_level3.c`, `proj2_nothread.c`, `prototype`, `prototype.c`, `pthread`, `read.txt`, `sample`, `sample.c`, `sample2.txt`, `sequence_of_commands.txt`, and `test.txt`. The main editor area has three tabs open: `commands.txt`, `read.txt`, and `empty.txt`, each containing the number '1'. At the bottom, a terminal window shows the following commands and output:

```
hans@DESKTOP-53529ML:~/CS_140/Proj2$ cat commands.txt
hans@DESKTOP-53529ML:~/CS_140/Proj2$ cat read.txt
hans@DESKTOP-53529ML:~/CS_140/Proj2$ cat empty.txt
hans@DESKTOP-53529ML:~/CS_140/Proj2$ ls
a.txt      empty.txt  file_server.c  proj2_level3.c  prototype      pthread      sample      sample2.txt  test
commands.txt  file_server  proj2         proj2_nothread.c  prototype.c    read.txt    sample.c    sequence_of_commands.txt  test.txt
```

Figure 11 test preliminaries

Testing level 1

The goal of testing for level 1 is to show that the master thread must continue taking input. There shall be no blocking. In order to demonstrate this, I will send multiple write requests in quick succession. Note that the files that I will write to do not exist yet but fopen will create these new files anyway. This was demonstrated better in the video documentation.

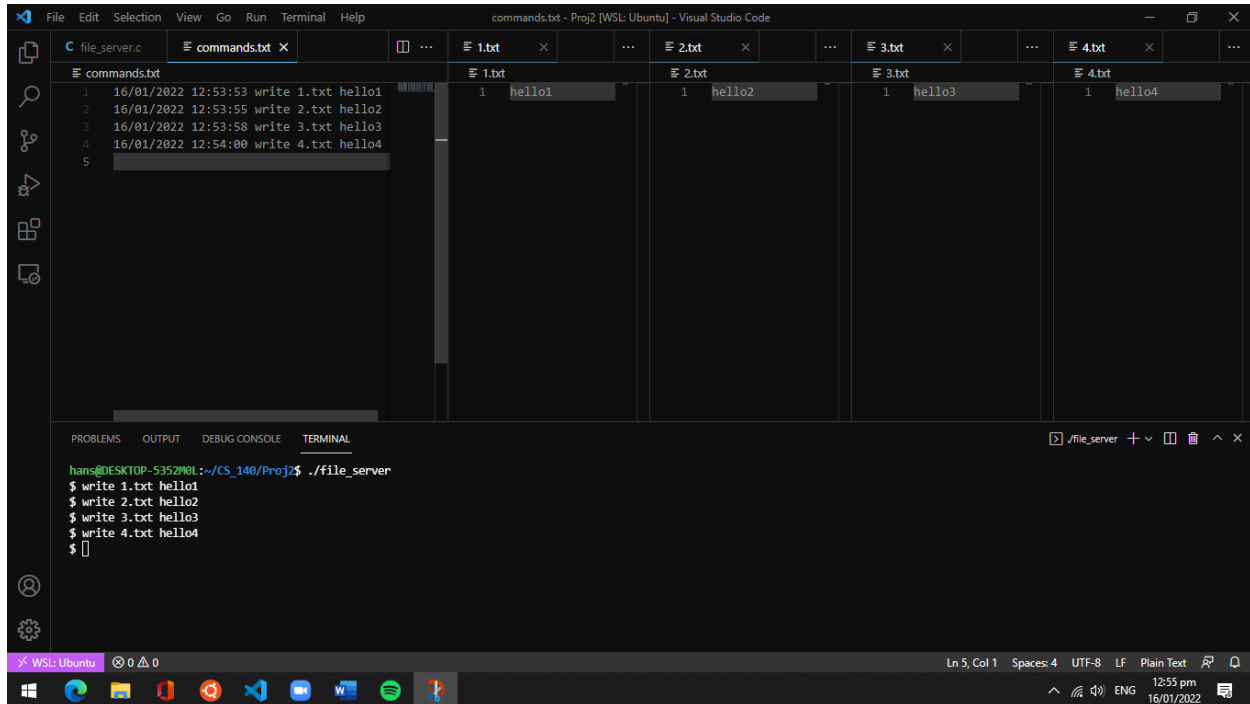


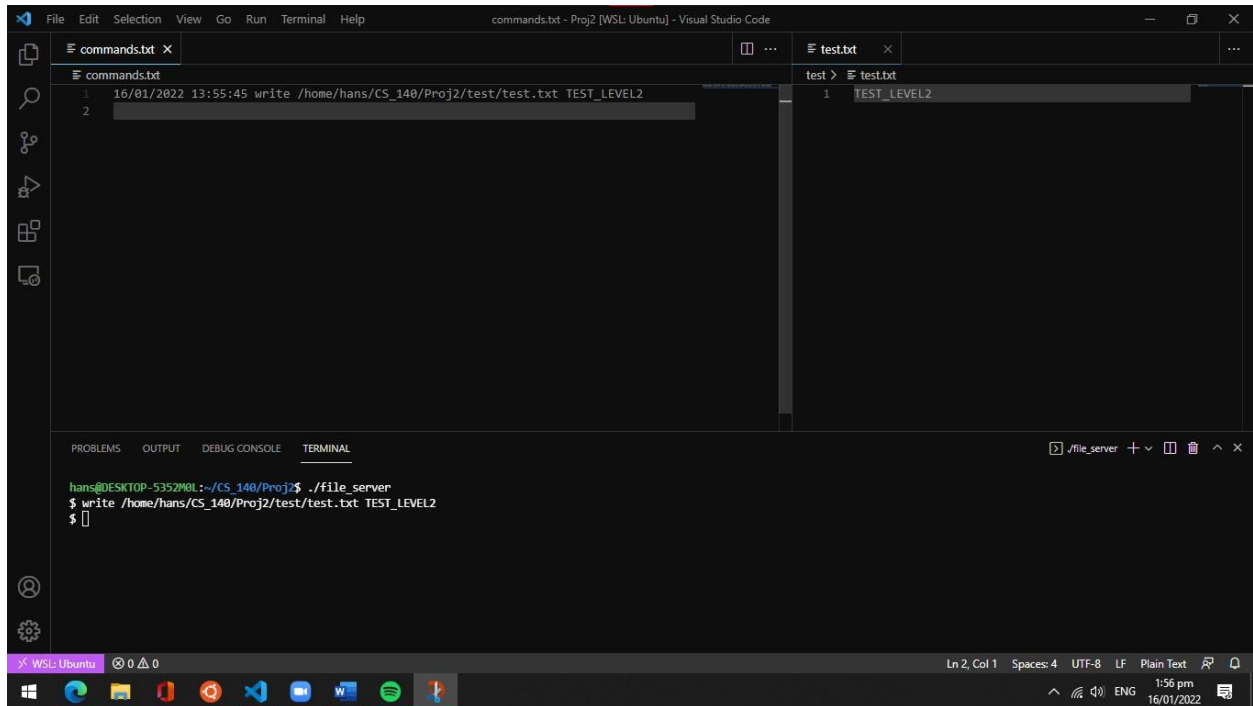
Figure 12 Level 1 test

We can see in figure 12 that write requests were called in quick succession. Although it is not entirely clear in a screenshot. We can see in commands.txt the timestamps when the requests were received. It seems that there is a 2 second difference between the inputs. Note that I tried my best to type very quickly. This test is much clear in the video.

Testing level 2

The goal of testing for level 2 is to show that file_server properly executes the entered commands. This means that the intended behaviors of write, read, and empty should reflect in files that they will create/modify. In the test below, note that the target file test.txt is in the directory /home/hans/CS_140/Proj2/test and it is initially empty.

Level 2 Testing write



The screenshot shows the Visual Studio Code interface with a dark theme. At the top, the menu bar includes File, Edit, Selection, View, Go, Run, Terminal, and Help. The title bar indicates the current workspace is 'commands.txt - Proj2 [WSL: Ubuntu] - Visual Studio Code'. The editor area has two tabs: 'commands.txt' and 'test.txt'. The 'commands.txt' tab is active, showing two lines of text: '1 16/01/2022 13:55:45 write /home/hans/CS_140/Proj2/test/test.txt TEST_LEVEL2' and '2'. The 'test.txt' tab is also visible, showing a single line of text: '1 TEST_LEVEL2'. Below the editor area is a panel with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is active, showing a shell prompt 'hans@DESKTOP-5352M0L:~/CS_140/Proj2\$./file_server' and the command '\$ write /home/hans/CS_140/Proj2/test/test.txt TEST_LEVEL2' followed by a prompt '\$ '. The status bar at the bottom shows 'Ln 2, Col 1', 'Spaces: 4', 'UTF-8', 'LF', 'Plain Text', and the date '16/01/2022'.

```
1 16/01/2022 13:55:45 write /home/hans/CS_140/Proj2/test/test.txt TEST_LEVEL2
2

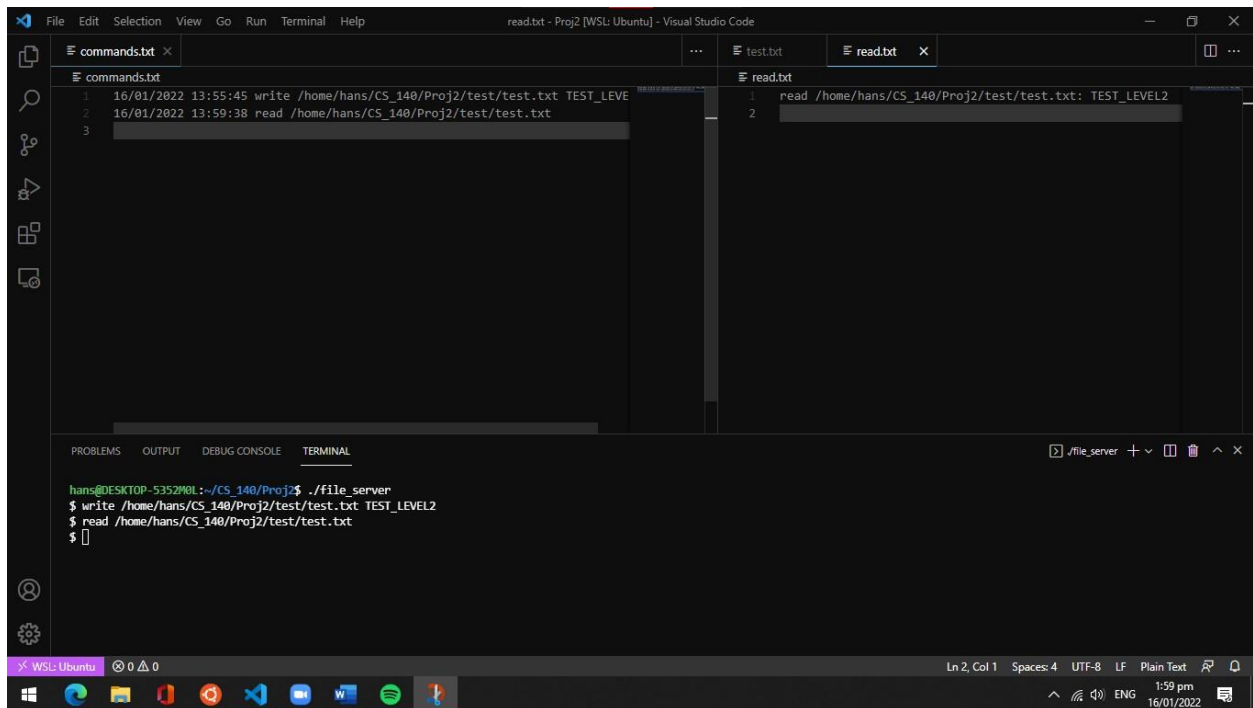
test > test.txt
1 TEST_LEVEL2

hans@DESKTOP-5352M0L:~/CS_140/Proj2$ ./file_server
$ write /home/hans/CS_140/Proj2/test/test.txt TEST_LEVEL2
$
```

Figure 13 Level 2 test write

We can see in Figure 13 that the command `write /home/hans/CS_140/Proj2/test/test.txt TEST_LEVEL2` worked. We can see the string “TEST_LEVEL2” written in test.txt on the right side. We can also see on the left that the entire command along with the timestamp is written into commands.txt

Level 2 Testing read



The screenshot displays the Visual Studio Code interface with a dark theme. The top editor pane shows the 'commands.txt' file with the following content:

```
1 16/01/2022 13:55:45 write /home/hans/CS_140/Proj2/test/test.txt TEST_LEVEL2
2 16/01/2022 13:59:38 read /home/hans/CS_140/Proj2/test/test.txt
3
```

The right editor pane shows the 'read.txt' file with the following content:

```
1 read /home/hans/CS_140/Proj2/test/test.txt: TEST_LEVEL2
2
```

The bottom panel shows the 'TERMINAL' tab with the following output:

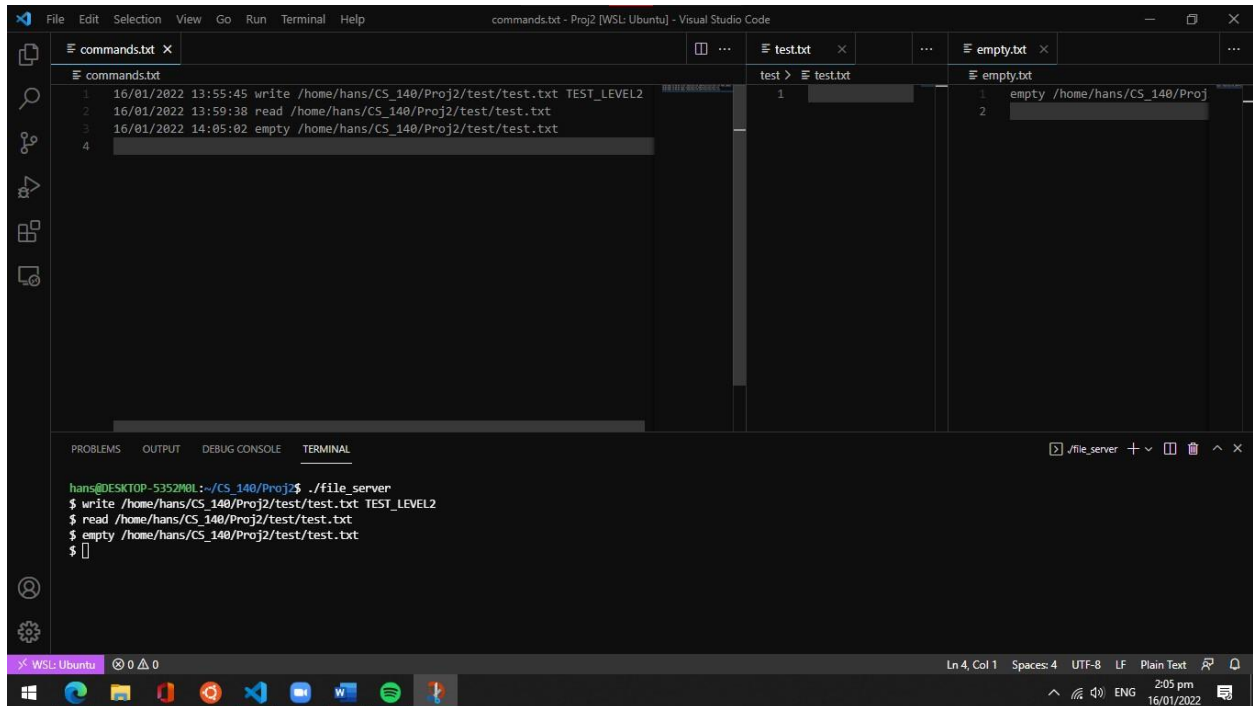
```
hans@DESKTOP-5352PHL:~/CS_140/Proj2$ ./file_server
$ write /home/hans/CS_140/Proj2/test/test.txt TEST_LEVEL2
$ read /home/hans/CS_140/Proj2/test/test.txt
$
```

The status bar at the bottom indicates the current file is 'Ln 2, Col 1' with 'Spaces: 4', 'UTF-8', 'LF', and 'Plain Text' encoding. The system tray shows the time as 1:59 pm on 16/01/2022.

Figure 14 Level 2 test read

We can see in the figure above that the content of test.txt was logged in read.txt. We can see TEST_LEVEL2 in read.txt. In addition to that, we can see that the read command is logged in commands.txt.

Level 2 Testing empty



```
File Edit Selection View Go Run Terminal Help
commands.txt - Proj2 [WSL: Ubuntu] - Visual Studio Code

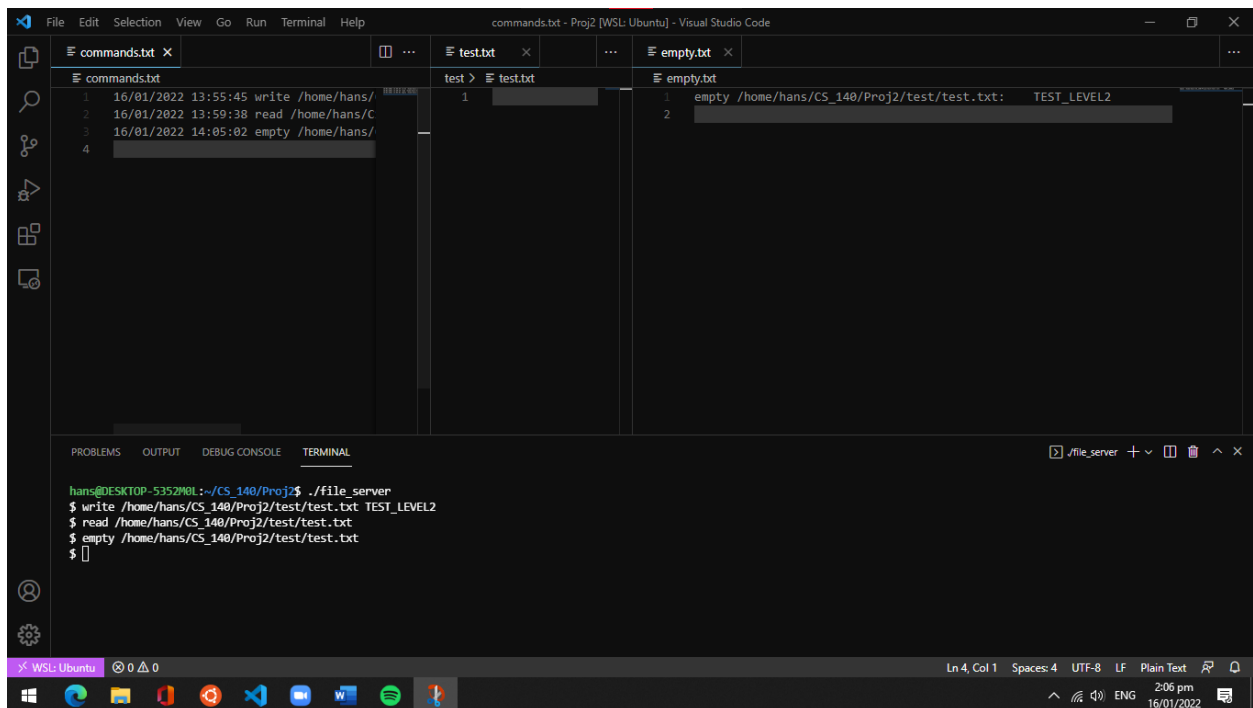
commands.txt
1 16/01/2022 13:55:45 write /home/hans/CS_140/Proj2/test/test.txt TEST_LEVEL2
2 16/01/2022 13:59:38 read /home/hans/CS_140/Proj2/test/test.txt
3 16/01/2022 14:05:02 empty /home/hans/CS_140/Proj2/test/test.txt
4

test.txt
1

empty.txt
1 empty /home/hans/CS_140/Proj2/test/test.txt:
2

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
hans@DESKTOP-5352M0L:~/CS_140/Proj2$ ./file_server
$ write /home/hans/CS_140/Proj2/test/test.txt TEST_LEVEL2
$ read /home/hans/CS_140/Proj2/test/test.txt
$ empty /home/hans/CS_140/Proj2/test/test.txt
$
```

Figure 15 Level 2 test 1 empty 1



```
File Edit Selection View Go Run Terminal Help
commands.txt - Proj2 [WSL: Ubuntu] - Visual Studio Code

commands.txt
1 16/01/2022 13:55:45 write /home/hans/
2 16/01/2022 13:59:38 read /home/hans/C
3 16/01/2022 14:05:02 empty /home/hans/
4

test.txt
1

empty.txt
1 empty /home/hans/CS_140/Proj2/test/test.txt: TEST_LEVEL2
2

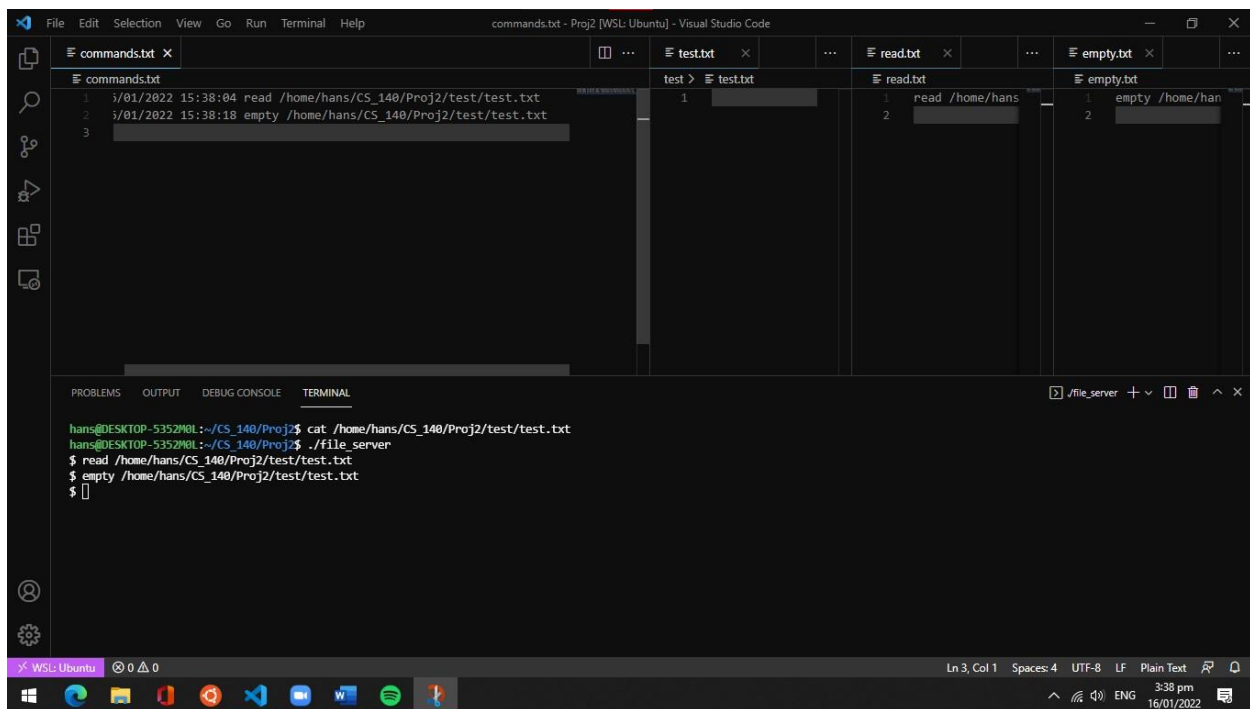
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
hans@DESKTOP-5352M0L:~/CS_140/Proj2$ ./file_server
$ write /home/hans/CS_140/Proj2/test/test.txt TEST_LEVEL2
$ read /home/hans/CS_140/Proj2/test/test.txt
$ empty /home/hans/CS_140/Proj2/test/test.txt
$
```

Figure 16 Level 2 test 1 empty 2

We can see in Figure 15 that the empty command was successfully logged into commands.txt. We can also see there that test.txt is now empty. It no longer has TEST_LEVEL2. In figure 16, we can now see the previous content of test.txt which is TEST_LEVEL2.

Level 2 testing empty files

For this second test in level 2, this sequence of commands will be ran: (1) read /home/hans/CS_140/Proj2/test/test.txt and (2) empty /home/hans/CS_140/Proj2/test/test.txt. Note that initially, commands.txt, read.txt, empty.txt, and test.txt are all empty. We expect to see 'FILE DNE' and 'FILE ALREADY EMPTY' in read.txt and empty.txt respectively.



```
File Edit Selection View Go Run Terminal Help
commands.txt - Proj2 [WSL: Ubuntu] - Visual Studio Code

commands.txt
1 i/01/2022 15:38:04 read /home/hans/CS_140/Proj2/test/test.txt
2 i/01/2022 15:38:18 empty /home/hans/CS_140/Proj2/test/test.txt
3

test.txt
1

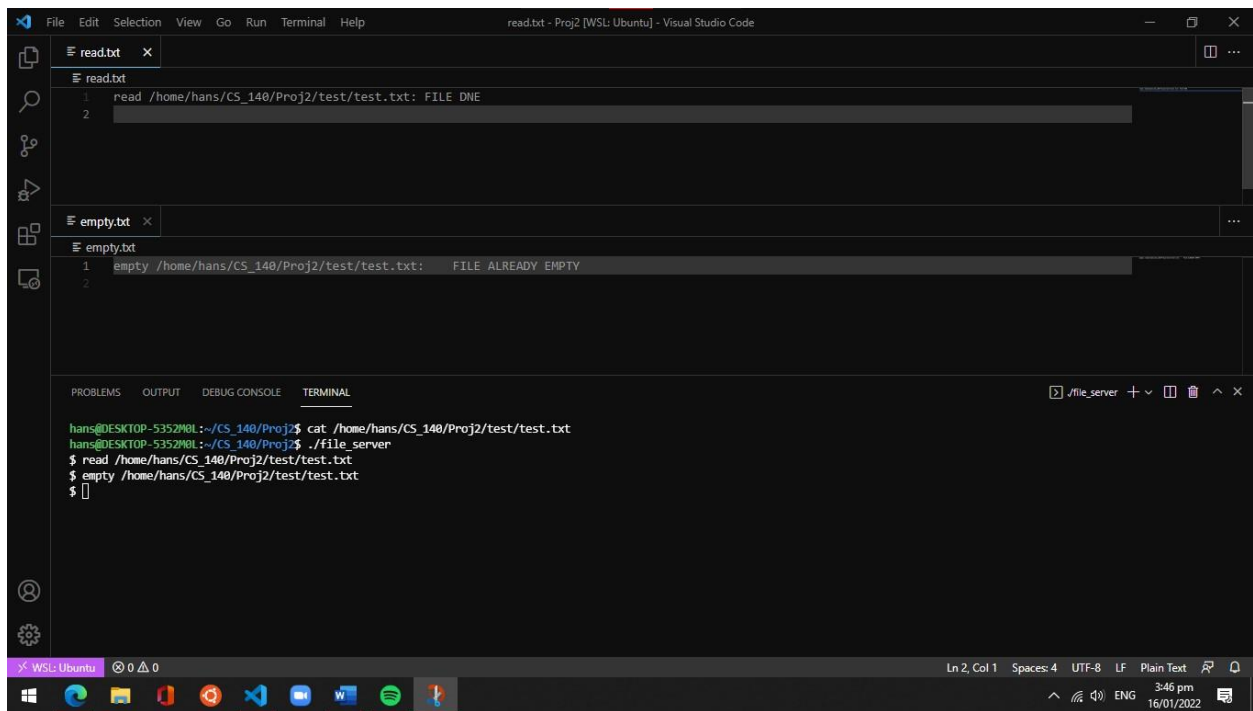
read.txt
1 read /home/hans
2

empty.txt
1 empty /home/han
2

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
hans@DESKTOP-53529BL:~/CS_140/Proj2$ cat /home/hans/CS_140/Proj2/test/test.txt
hans@DESKTOP-53529BL:~/CS_140/Proj2$ ./file_server
$ read /home/hans/CS_140/Proj2/test/test.txt
$ empty /home/hans/CS_140/Proj2/test/test.txt
$
```

Figure 17 Level 2 test empty files 1

We can see in figure 17 that the issued commands were successfully logged into commands.txt. We can also see that test.txt is empty. This can also be proven by the cat /home/hans/CS_140/Proj2/test/test.txt command in the terminal below. It outputs nothing in the terminal before ./file_server.



The screenshot shows the Visual Studio Code interface with two open text files and a terminal window. The top file, 'read.txt', contains two lines of code: `1 read /home/hans/CS_140/Proj2/test/test.txt: FILE DNE` and `2`. The bottom file, 'empty.txt', contains two lines of code: `1 empty /home/hans/CS_140/Proj2/test/test.txt: FILE ALREADY EMPTY` and `2`. The terminal window at the bottom shows the following commands and output:

```
hans@DESKTOP-5352M8L:~/CS_140/Proj2$ cat /home/hans/CS_140/Proj2/test/test.txt
hans@DESKTOP-5352M8L:~/CS_140/Proj2$ ./file_server
$ read /home/hans/CS_140/Proj2/test/test.txt
$ empty /home/hans/CS_140/Proj2/test/test.txt
$
```

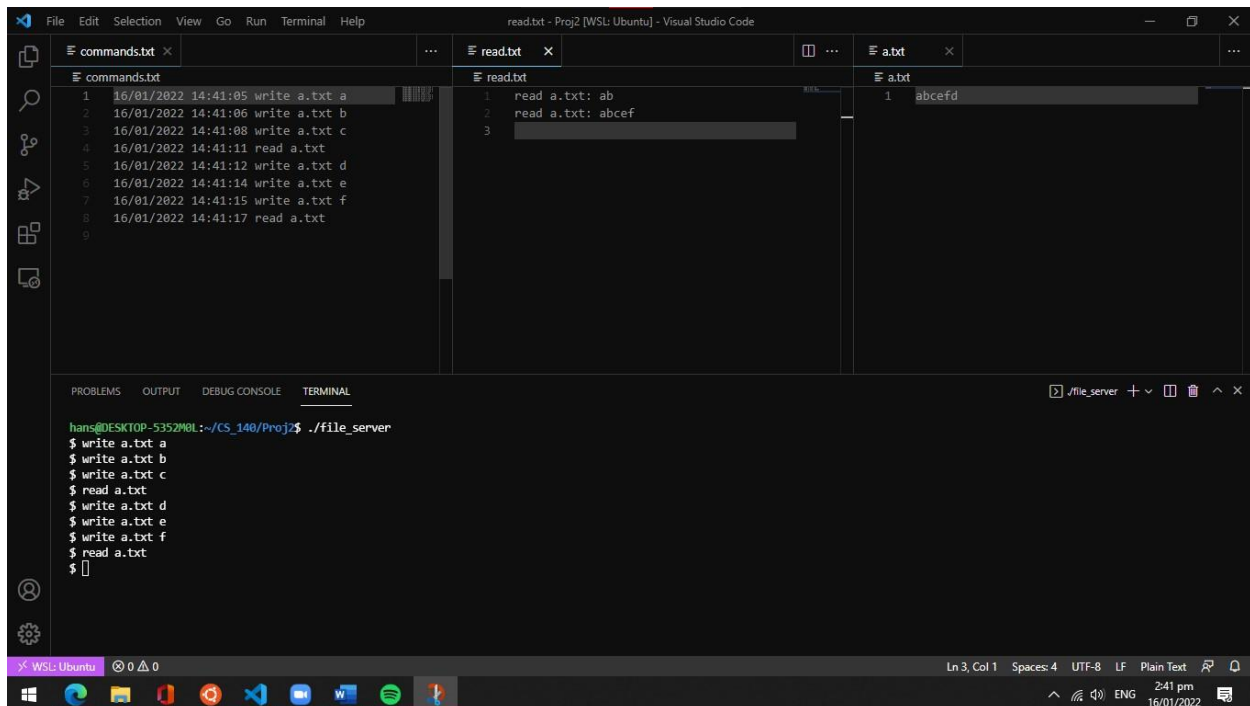
The status bar at the bottom indicates the current file is 'Ln 2, Col 1', using 'Spaces: 4', 'UTF-8' encoding, 'LF' line endings, and 'Plain Text' format. The system clock shows 3:46 pm on 16/01/2022.

Figure 18 Level 2 test empty files 2

We can see in figure 18 that the appropriate responses for reading and empty file were made. We can see 'FILE DNE' in read.txt and 'FILE ALREADY EMPTY' in empty.txt.

Testing Level 3

The goal of testing for level 3 is to show that file_server is able to worker threads are executing at the same time. In order to demonstrate this, I will run this sequence of commands in quick succession: (1) write a.txt a, (2) write a.txt b, (3) write a.txt c, (4) read a.txt, (5) write a.txt d, (6) write a.txt e, (7) write a.txt f, (8) read a.txt. If by luck, the threads run in sequence, we will expect to see 'abc' in read.txt after (4) and 'abcdef' in read.txt after (8). If it does not look like those two, then that is a hint that the threads might be running at the same time. Note that a.txt will be in the same directory as file_server so no need to specify its full path.



```
File Edit Selection View Go Run Terminal Help
read.txt - Proj2 [WSL: Ubuntu] - Visual Studio Code

commands.txt
1 16/01/2022 14:41:05 write a.txt a
2 16/01/2022 14:41:06 write a.txt b
3 16/01/2022 14:41:08 write a.txt c
4 16/01/2022 14:41:11 read a.txt
5 16/01/2022 14:41:12 write a.txt d
6 16/01/2022 14:41:14 write a.txt e
7 16/01/2022 14:41:15 write a.txt f
8 16/01/2022 14:41:17 read a.txt
9

read.txt
1 read a.txt: ab
2 read a.txt: abcef
3

a.txt
1 abcefd

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
hans@DESKTOP-5352M8L:~/CS_140/Proj2$ ./file_server
$ write a.txt a
$ write a.txt b
$ write a.txt c
$ read a.txt
$ write a.txt d
$ write a.txt e
$ write a.txt f
$ read a.txt
$
```

Figure 19 Level 3 test 1

We said earlier that we expect to see 'abc' in read.txt right after the first (4) read a.txt. But instead, we see only 'ab'. We also said that we expect to see 'abcdef' in read.txt right after the second (8) read a.txt. However, looking at figure 19, we see 'abcef' instead. Finally, we can see 'abcefd' in a.txt. We can infer from this that all the threads spawned did not wait for each other to finish before executing. We can see that 'd' is appended last even though its request was earlier than write a.txt e and write a.txt f.

For this second test for level 3, we will be using the empty command. I will be running this sequence of commands: (1) empty a.txt, (2) read a.txt, (3) write a.txt level3, (4) read a.txt. We assume that a.txt contains 'abcefd' initially, command.txt, read.txt, and empty.txt are all empty. We expect to see 'FILE DNE' in read.txt after (2) because there was an empty a.txt before it. We also expect to see 'level3' in read.txt after (4) because of (3).

```
File Edit Selection View Go Run Terminal Help
a.txt - Proj2 [WSL: Ubuntu] - Visual Studio Code

commands.txt x a.txt x read.txt x empty.txt x
1 16/01/2022 15:17:43 empty a.txt
2 16/01/2022 15:17:45 read a.txt
3 16/01/2022 15:17:48 write a.txt level3
4 16/01/2022 15:17:49 read a.txt
5

1 level3

1 read a.txt: FILE DNE
2 read a.txt: level3
3

1 empty a.txt: abcefd
2

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
hans@DESKTOP-5352M0L:~/CS_140/Proj2$ cat a.txt
abcefdhans@DESKTOP-5352M0L:~/CS_140/Proj2$ ./file_server
$ empty a.txt
$ read a.txt
$ write a.txt level3
$ read a.txt
$
```

Figure 20 Level 3 test 2

First things first, we can see in the terminal in figure 20 that the result of `cat a.txt` is `abcefd`. This just proves that `a.txt` initially contains 'abcefd'. Next, we see 'FILE DNE' in the first `read a.txt` in `read.txt`. It might be possible that while `empty a.txt` is still running but has already finished emptying `a.txt`, the first `read a.txt` now sees an empty `a.txt` even though `empty a.txt` might not have terminated yet because it is expected to sleep for 7-10 seconds after emptying it. This proves that the threads are not waiting for each other to finish before execution. The example above also shows that it is entirely possible that `empty a.txt` only slept for 1 s (this is most likely because its probability is 80%) because we can see that the time difference between `empty a.txt` and the first `read a.txt` is only around 2 seconds (sorry that's just the fastest that I typed the commands).

Video Documentation Link

[CS 140 Proj 2 - 201903184.mp4](#)