Achieving Concurrency

The screenshot below portrays the program before any changes were made to achieve concurrency. In this server source file, there is the definition of the server address, then we are binding the socket to our specified IP & port using an if statement, afterwards we are listening to any connections and eventually we connect to the client. Once we are connected our IoT_sensor starts working by calculating the MOISTURE, PH and SUNLIGHT.

```
//our two way connection
                client_socket = accept(server_socket,NULL, NULL);
                if (client_socket != -1) {

                        int count = 1;

                        while(count == 1){
                                memset(str, 0, MAX_STR);

                                if(read(client_socket,str, MAX_STR-1) == 0){
                                        count++;

                                }else{

                                        FILE* fp;

                                        fp = fopen("file.dat", "rb");

                                        if(!fp || fp == NULL){
                                                printf("error reading file");
                                                fp =fopen("file.dat", "wb");
                                        }else{

                                                fread(&s, sizeof(soil), 1, fp);
                                                printf("Moisture : %d\n",s.c_moi);
                                                printf("PH : %d\n",s.c_ph);
                                                printf("SUNLIGHT : %d\n",s.c_sun);

                                        }
                                        fclose(fp);


                                        //sprintf >> takes the integer and stores it as a string.
                                        if(strncmp(str, "MOISTURE", 8) == 0){
                                                fp = fopen("file.dat", "wb+");
                                                fread(&s, sizeof(soil), 1, fp);
                                                num = (rand()%10+1);
                                                sprintf(str, "%d ", num);
                                                s.c_moi++;
                                                printf("%s\n", str);
                                                fwrite(&s.c_moi, sizeof(soil), 1, fp);
                                                fclose(fp);

                                        }else if(strncmp(str, "PH", 2) == 0){
                                                fp = fopen("file.dat", "wb+");
                                                fread(&s, sizeof(soil), 1, fp);
                                                num = (rand()%10+1);
                                                sprintf(str, "%d ", num);
                                                s.c_ph++;
                                                fwrite(&s, sizeof(soil), 1, fp);
                                                fclose(fp);
```

Amber Debono
SWD 6.2A

```
if(strncmp(str, "MOISTURE", 8) == 0){
        fp = fopen("file.dat", "wb+");
        fread(&s, sizeof(soil), 1, fp);
        num = (rand()%10+1);
        sprintf(str, "%d ", num);
        s.c_moi++;
        printf("%s\n", str);
        fwrite(&s.c_moi, sizeof(soil), 1, fp);
        fclose(fp);

}else if(strncmp(str, "PH", 2) == 0){
        fp = fopen("file.dat", "wb+");
        fread(&s, sizeof(soil), 1, fp);
        num = (rand()%10+1);
        sprintf(str, "%d ", num);
        s.c_ph++;
        fwrite(&s, sizeof(soil), 1, fp);
        fclose(fp);

}else if(strncmp(str, "SUNLIGHT", 8) == 0){
        fp = fopen("file.dat", "wb+");
        fread(&s, sizeof(soil), 1, fp);
        int num2 = (rand()%2000+1);
        sprintf(str, "%d ", num2);
        s.c_sun++;
        fwrite(&s, sizeof(soil), 1, fp);
        fclose(fp);

}else if(strncmp(str, "STATS", 5) == 0){
        sprintf(str, "%d MOISTURE, %d PH, %d SUNLIGHT\n", s.c_moi, s.c_ph, s.c_sun);

}else if(strncmp(str, "RESET", 5) == 0){
        //Everytime the client writes reset, the file will be emptied
        fp = fopen("file.dat", "wb");

        memset(&s, 0, sizeof(s));
        sprintf(str, "%d MOISTURE, %d PH, %d SUNLIGHT\n", s.c_moi, s.c_ph, s.c_sun);

        fclose(fp);
}else{
        printf( "Unknown \n");
}

printf("Number: %s\n", str);
write(client_socket, str, strlen(str)+1);
```

Moving on to part C in the assignment, as it was instructed there was a restructure of our code to achieve concurrency. The definition of the server address, the bind and the listen function are still in the server source file. The initialization and destroy of the mutex are also included in the server source file. After listening to connections there is a while loop that when is true, we start accepting connections. However, before we do so, there is a change in the client_socket variable where instead of leaving it as a variable, it is changed to a pointer so when using the pthread_create(), we can pass a void pointer. Once it is connected, a thread is a created and afterwards we will use the pthread_create(), where we pass the address of our thread, a NULL, the function which we will be using and lastly the argument passed onto the function which in this case is the client_socket pointer. Validations are made such as checking if malloc is null and if the client connection is accepted. Finally, there is a label referred to as cleanup_server to deduct redundant code.

```
    //initializing my mutex
    pthread_mutex_init(&mutex, NULL);
```

```c
int *p_client_socket = malloc(sizeof(int));

//Checking if malloc is null
if (p_client_socket == NULL)  {
        fprintf(stderr, "ERROR: malloc() is NULL\n");
        goto cleanup_server;
}

//our two way connection
*p_client_socket = accept(server_socket,(struct sockaddr *)&client_address, &clilen);


if (*p_client_socket == -1) {
        fprintf(stderr, "ERROR: accept()\n");
        goto cleanup_server;
}

printf(" -> New Client\n");

pthread_t thread;

//Passing the address of our thread,No attributes=NULL, the function, the argument passed onto the function-> saving the response of the client
pthread_create(&thread, NULL, handle_connection, p_client_socket);
```

```c
pthread_mutex_destroy(&mutex);

cleanup_server:
close(server_socket);
return -2;
```

The function is then created in the header file and called from the server source file in the pthread_create(). Moving onto the header file, where the function handle_connection is created. As mentioned previously, the parameter being passed has to be a pointer, therefore *p_client socket is passed and later on it is type-casted and freed immediately since there is no use for it. Also, the data structure has been moved to the header file due to its use being in this file. Moreover, a loop is created to wait for client connection and if the client succeeds the program enter in the else section. Primarily a file declared and before it is opened it is being locked. The locks are put in every location where a file is being opened and an unlock when it is closed. The reason being is that we do not want a thread to interrupt another thread. Afterwards, there is a check whether the file exists or not, and if it does not exit it will be created immediately.

```c
FILE* fp;

pthread_mutex_lock(&mutex);

fp = fopen("file.dat", "rb");

if(!fp || fp == NULL){
    printf("error reading file");
    fp =fopen("file.dat", "wb");
    memset(&s, 0, sizeof(s));
    fclose(fp);
}
fclose(fp);
pthread_mutex_unlock(&mutex);
```

Furthermore, in the if statements that come after, we are first checking what the user inputted out of the five commands and if it matches it will go into a particular if. These statements where first in the server source file and now moved into the header file with a few changes. Starting of with putting a lock on the file so no other client can access, we are then opening the file in a read binary mode. We are reading from the temporary struct which contains the whole struct that was previously saved. We are then choosing the particular command that was chosen and put it in our main struct and later on incrementing it to update our STATS. Proceeding to close the file, we open it again but this time in write mode to write in the file our updated struct with the incremented command. In conclusion, the file is closed and now we can unlock.

```
if(strncmp(str, "MOISTURE", 8) == 0){
    pthread_mutex_lock(&mutex);
     fp = fopen("file.dat", "rb");
    fread(&temp, sizeof(soil), 1, fp);
    s.c_moi = temp.c_moi;
    num = (rand()%10+1);
    sprintf(str, "%d ", num);
    s.c_moi++;
     fclose(fp);
    printf("%s\n", str);
    fp = fopen("file.dat", "wb");
    fwrite(&s, sizeof(soil), 1, fp);
    fclose(fp);
    pthread_mutex_unlock(&mutex);

}else if(strncmp(str, "PH", 2) == 0){
    pthread_mutex_lock(&mutex);
    fp = fopen("file.dat", "rb");
    fread(&temp, sizeof(soil), 1, fp);
    s.c_ph = temp.c_ph;
    num = (rand()%10+1);
    sprintf(str, "%d ", num);
    s.c_ph++;
    fclose(fp);
    printf("%s\n", str);
    fp = fopen("file.dat", "wb");
    fwrite(&s, sizeof(soil), 1, fp);
    fclose(fp);
    pthread_mutex_unlock(&mutex);

}else if(strncmp(str, "SUNLIGHT", 8) == 0){
    pthread_mutex_lock(&mutex);
    fp = fopen("file.dat", "rb");
    fread(&temp, sizeof(soil), 1, fp);
    s.c_sun = temp.c_sun;
    int num2 = (rand()%2000+1);
    sprintf(str, "%d ", num2);
    s.c_sun++;
    fclose(fp);
    printf("%s\n", str);
    fp = fopen("file.dat", "wb");
    fwrite(&s, sizeof(soil), 1, fp);
    fclose(fp);
    pthread_mutex_unlock(&mutex);

}else if(strncmp(str, "STATS", 5) == 0){
    pthread_mutex_lock(&mutex);
    fp = fopen("file.dat", "rb");
    fread(&s, sizeof(soil), 1, fp);
    sprintf(str, "%d MOISTURE, %d PH, %d SUNLIGHT\n", s.c_moi, s.c_ph, s.c_sun);
    fclose(fp);
```