# Gebze Technical University

## Department Of Computer Engineering

## CSE 344 Spring 2023

## System Programming

Final

Due Date: 16.06.2023

Abdurrahman Bulut
1901042258

# Introduction

In this project, I implemented a simplified version of Dropbox. The server side can handle multiple clients simultaneously and synchronizes directories with clients. The server also maintains a log file for each client and handles SIGINT signals.

To implement these requirements, I created a server and client program. The server program listens for connections from clients, and the client program connects to the server and synchronizes its directory with the server. I implemented the synchronization feature by creating a thread for each client and having each thread handle file operations on the server and client sides. I implemented the log file feature by creating a log file for each client and writing the names and access times of created, removed, and updated files to the logfile.

In my program, when clients connect to the Server, they will be sync immediately. After adding something to the clients, it will be added to the Server directory also. What I was missing was that I couldn't handle file or folder deleting. Actually it was working but I broke the code.. Another thing, I am sending log files also. They shouldn't be transferred to the clients. I should keep the log files in the clients directories but I am keeping them in the server folder. Thread mechanism is working well on both sides. I am using another thread on the client side for inotify.

Finally, I implemented the SIGINT signal handler by gracefully shutting down the server and client programs when the SIGINT signal is received.

I have submitted my source files, my makefile, my readMe file, and my report. My makefile only compiles the program, and my report is in PDF format.

# Implementation

## **Server**:

```
// BAĞLANAN CLİENT LİSTESİ
connectedClients = (ModifiedFile*)malloc(4096*sizeof(ModifiedFile));


int *numbers = (int*)malloc(threadPoolSize*sizeof(int));
for (int i = 0; i < threadPoolSize; ++i){
    numbers[i]=i;
}

clientFD = (int*)malloc(threadPoolSize*sizeof(int));
serverThreads = (pthread_t *)malloc(threadPoolSize * sizeof(pthread_t));


for (int i = 0; i < threadPoolSize; ++i){
    clientFD[i] = -1;

    // İŞLEM BURDA YAPILIYOR
    int err = pthread_create(&serverThreads[i], NULL, handleClient, (void*)&numbers[i]);
    if(err) threadErr(err,"thread creation");
}

for (int i = 0; i < threadPoolSize; ++i){
    int err = pthread_join (serverThreads[i], NULL);
    if(err) threadErr(err,"thread join");
}
```

The main function of the code is to create a server that listens for connections from clients. The server will then create a thread for each client that connects, and the thread will handle the client's requests. The code first checks to make sure that the user has provided the correct number of arguments. If the number of arguments is incorrect, the code prints an error message and exits.

Next, the code gets the thread pool size, port number, and server path from the command line arguments. The thread pool size is the number of threads that will be created to handle client requests. The port number is the port that the server will listen on for connections. The server path is the path to the directory that the server will serve files from. The code then sets up a signal handler for SIGINT and SIGTERM signals. These signals are sent when the user presses Ctrl+C or kills the process. The signal handler will gracefully shut down the server when one of these signals is received.

The code then creates a server socket and binds it to the specified port. The server socket is then put into listen mode, which means that it will accept connections from clients. The code then creates a linked list of ModifiedFile structures. These structures

will be used to keep track of the files that have been modified by clients. The code then creates an array of integers and a pthread_t array. The integers will be used to pass the thread IDs to the handleClient function. The pthread_t array will be used to store the thread handles. The code then enters a loop that creates a thread for each client that connects to the server. The handleClient function will be executed in each thread. The handleClient function will handle the client's requests. The code then enters a loop that waits for all of the threads to terminate. When all of the threads have terminated, the code closes the server socket and frees the allocated memory.

In line 171: The function handleClient is a thread function that handles a client connection. The function first reads the client's directory name, and then checks if the client is already connected. If the client is already connected, the function sends an error message to the client. Otherwise, the function adds the client's directory name to the list of online clients. The function then enters a loop that reads messages from the client. For each message, the function checks the message's flag. If the flag is 0, the function reads the file name and mode from the message, and then checks if the file exists. If the file exists, the function opens the file and writes the content of the message to the file. If the file does not exist, the function creates the file and writes the content of the message to the file. If the flag is 1, the function indicates that the client is finished sending messages. The function then closes the file if it is open, and then removes the client's directory name from the list of online clients.

Line 381: Actually I could not use this function properly. If it works fine, I can complete the deleting part but I couldn't. The function first opens the directory specified by the path. Then, the function iterates through the directory, calling itself recursively for any directories that are found. For each file that is found, the function checks if the file is in the list. If the file is not in the list, the function removes the file and writes a log message to the log file.

Signal Handler func: If CTRL+C is pressed, It will send a “shutdown” message to each client.

```
void signalHandler(int signo){
    FileEntry resp;
    memset(&resp, 0, sizeof resp);

    if(pthread_self() == mainThread){

        if(signo == SIGINT)
            printf("SIGINT handled.\n");
        else if(signo == SIGTERM)
            printf("SIGTERM handled.\n");

        for (int i = 0; i < threadPoolSize; ++i){
            if(clientFD[i]!=-1){
                strcpy(resp.content, "shutdown");
                write(clientFD[i], &resp, sizeof(FileEntry));
                memset(&resp, 0, sizeof resp);
            }
        }

        delay(1000);

        for (int i = 0; i < threadPoolSize; ++i){
            pthread_cancel(serverThreads[i]);
        }
    }

}
```

In line 514 and 528:

I actually fetched this function from the book. The function sendServerToClient takes a filename and a file descriptor as input. The function then sends the file to the client, one file at a time. The function first sends the file name to the client. Then, the function reads a response message from the client.

The function sendServerFiles takes a directory name and a file descriptor as input. The function then recursively sends all of the files in the directory to the client. The function first sends the directory name to the client. Then, the function reads a response message from the client.

## Client:

The main function in this program is responsible for initializing the client socket, connecting to the server, and sending the client directory name to the server. The function then creates a thread to monitor the client directory for changes, and enters a loop where it sends the client directory to the server and reads the server's response. The loop terminates when the user presses Ctrl+C.

I did a similar thing with the server side. The function readServer reads the files sent by the server and saves them to the client directory. The function first reads a FileEntry structure from the server. This structure contains the name of the file, its mode, and its flag. The flag indicates whether the file is a directory or a regular file.

If the file is a directory, the function recursively calls itself to read the files in the directory. If the file is a regular file, the function opens the file and reads its contents. The function then writes the file contents back to the client directory. The function continues to read files from the server until the server sends a FileEntry structure with a flag of 3. This indicates that the server has sent all of the files.

I used the “inotify” library to get feedback when a changes made. The function “MonitorDirectoryChangesRecursively” recursively monitors a directory for changes. The function first creates a watch descriptor for the directory. Then, it calls itself recursively for each subdirectory in the directory. The function “MonitorDirectoryChanges” creates a thread to monitor the directory for changes. The thread calls the MonitorDirectoryChangesRecursively function to recursively monitor the directory. The thread then enters a loop where it reads events from the inotify file descriptor. For each event, the thread checks the event type and takes appropriate action.

This is my thread structure to use inotify.

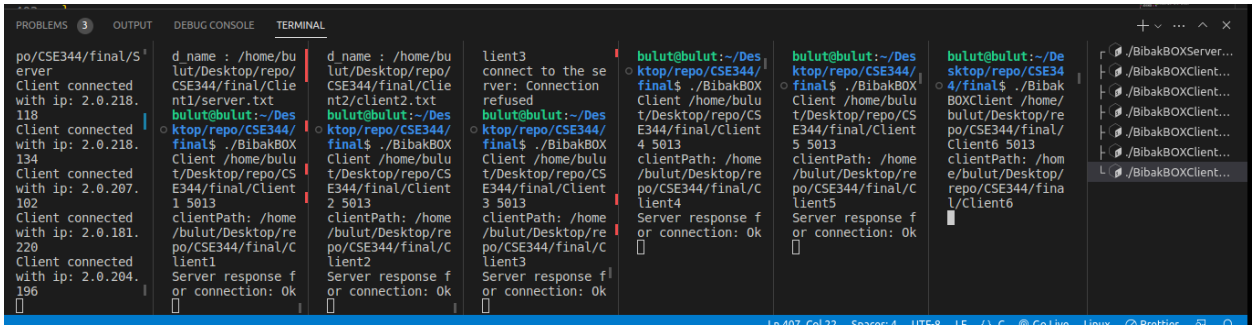
```c
void MonitorDirectoryChangesRecursively(const char *directoryPath, int fileDescriptor);
void MonitorDirectoryChanges(const char *directoryPath, int clientSocket);

void *threadFunction(void *arg) {
    ThreadParams *params = (ThreadParams *)arg;
    MonitorDirectoryChanges(params->directoryPath, params->clientSocket);
    return NULL;
}
```

# Tests

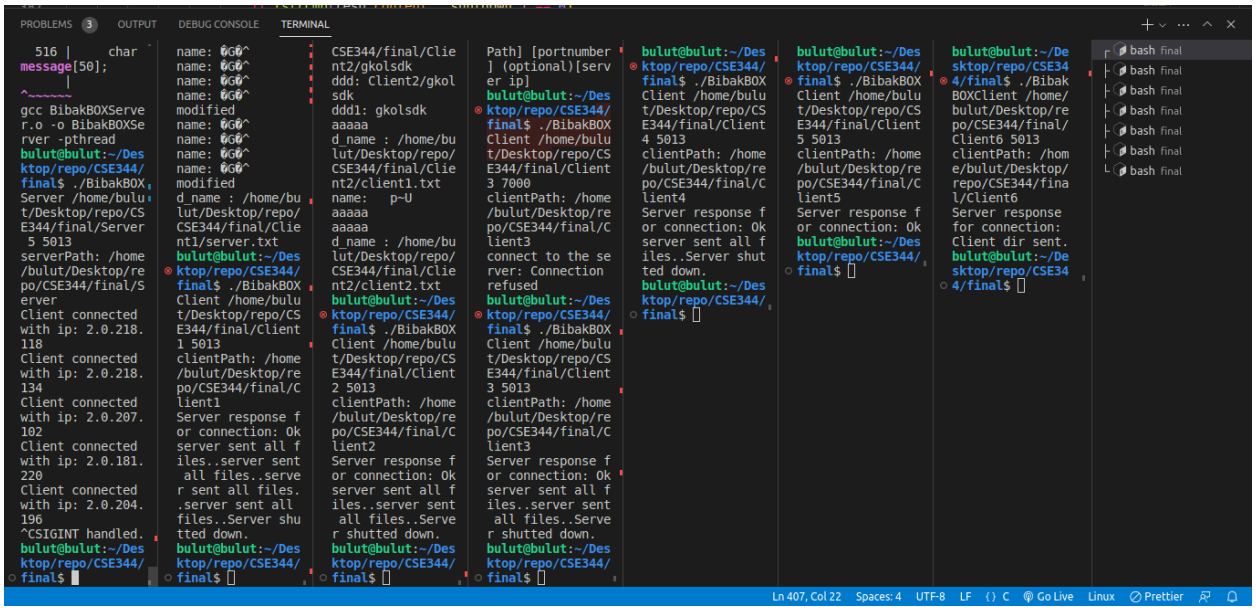
## Client connects:

When I connect multiple clients.

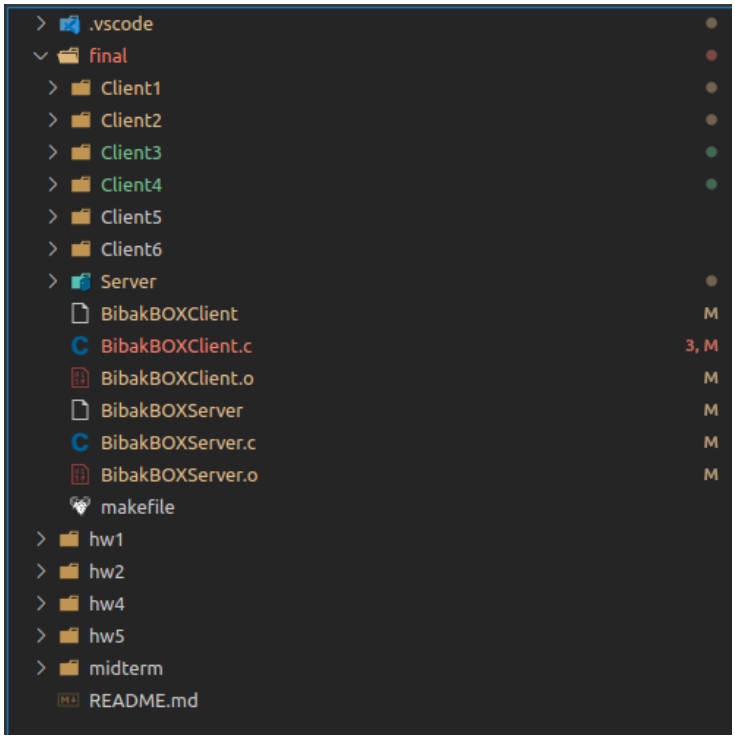



## SigInt test:

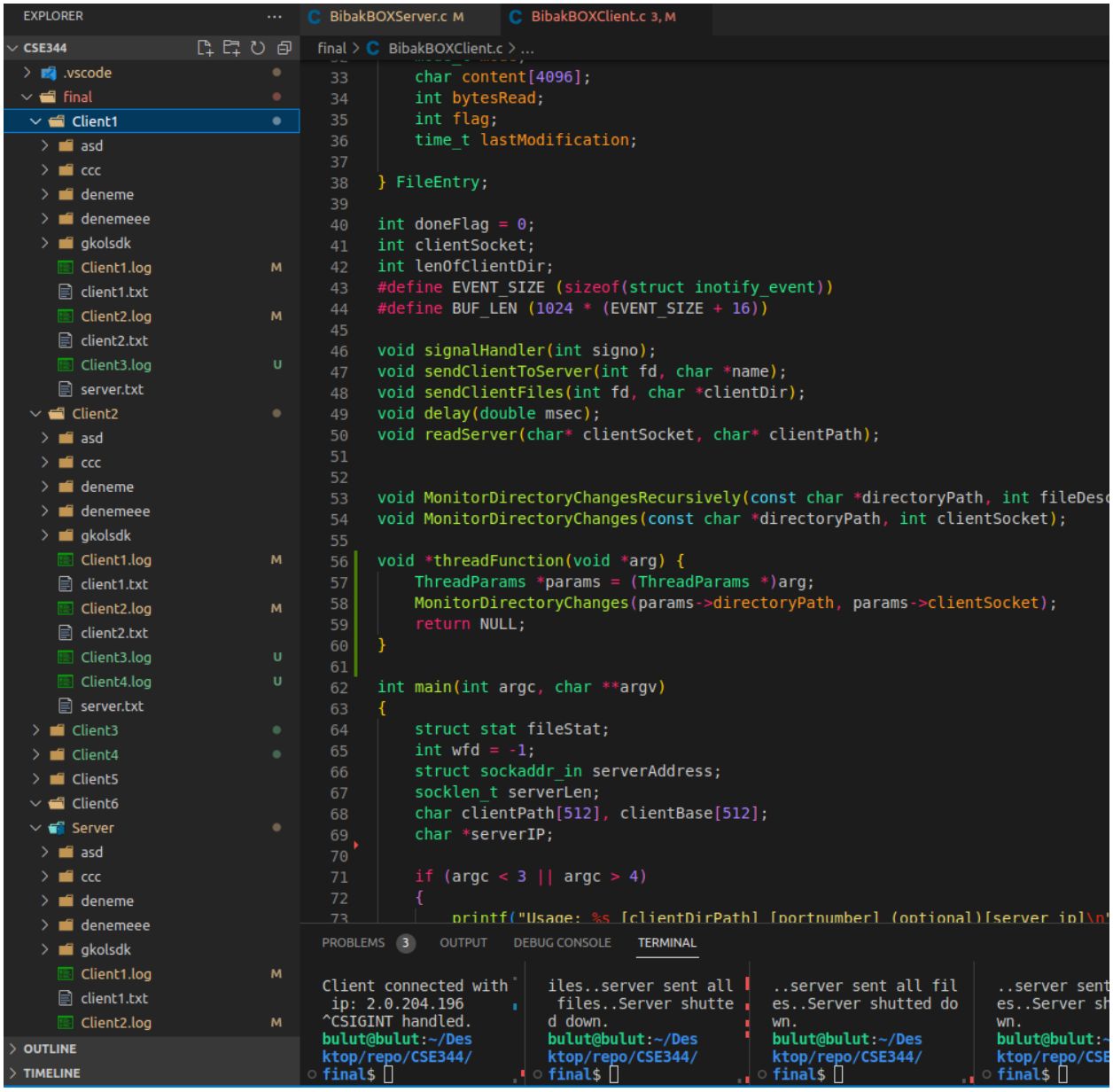
When I click ctrl+c on the server side.



My clients and server.

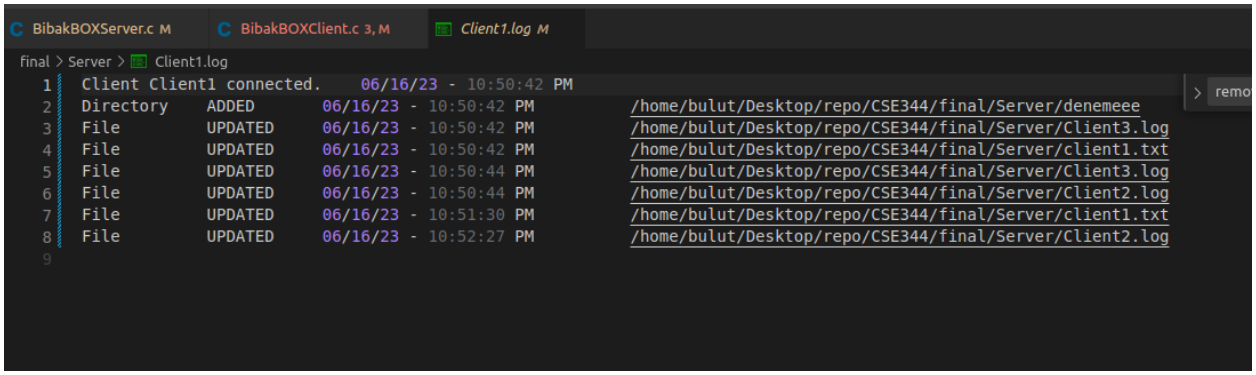


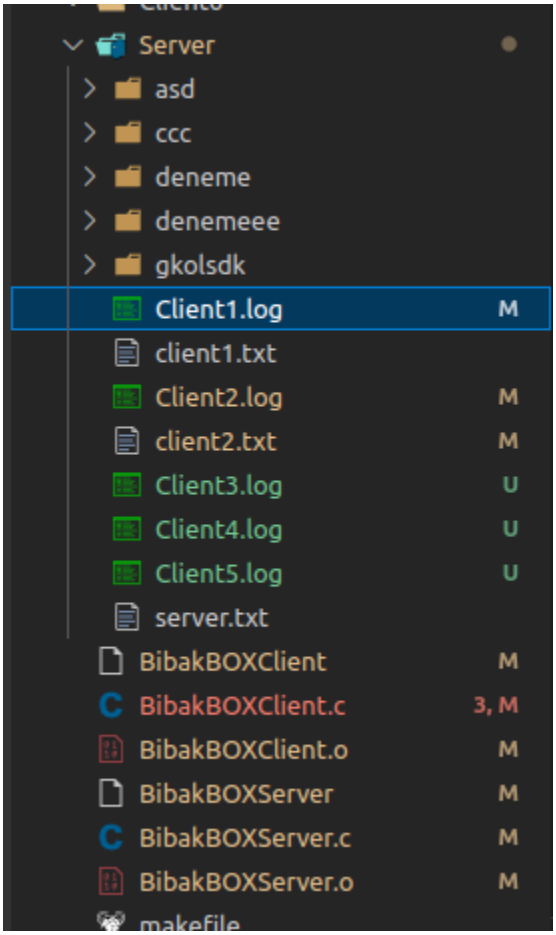
When a client connects to the server, all files and folders are syncronized in time.

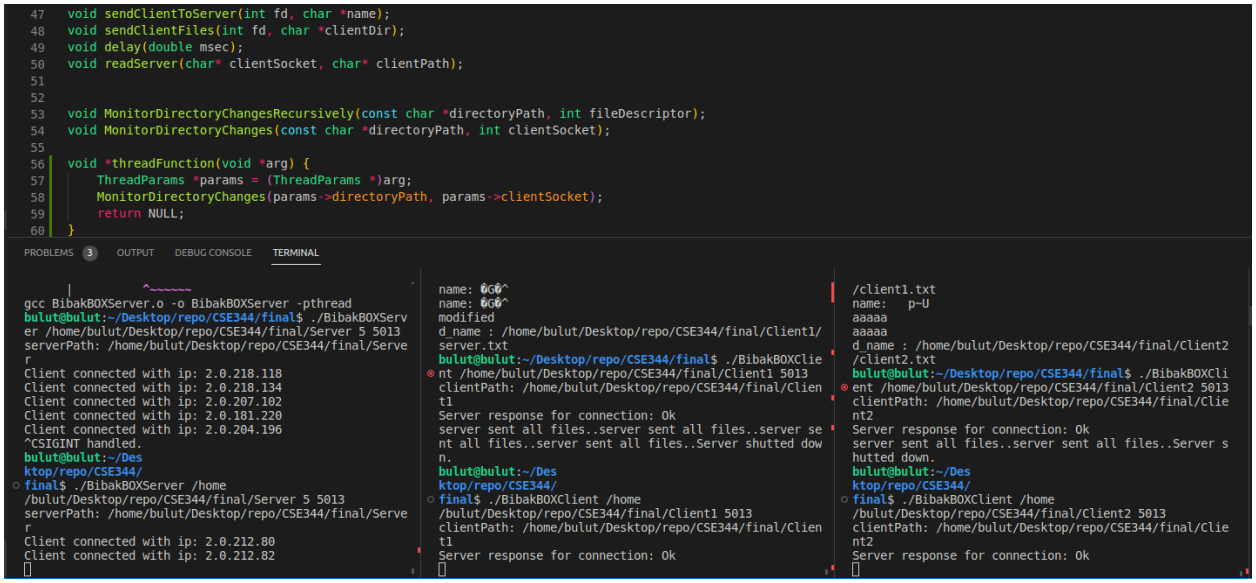


Log files:

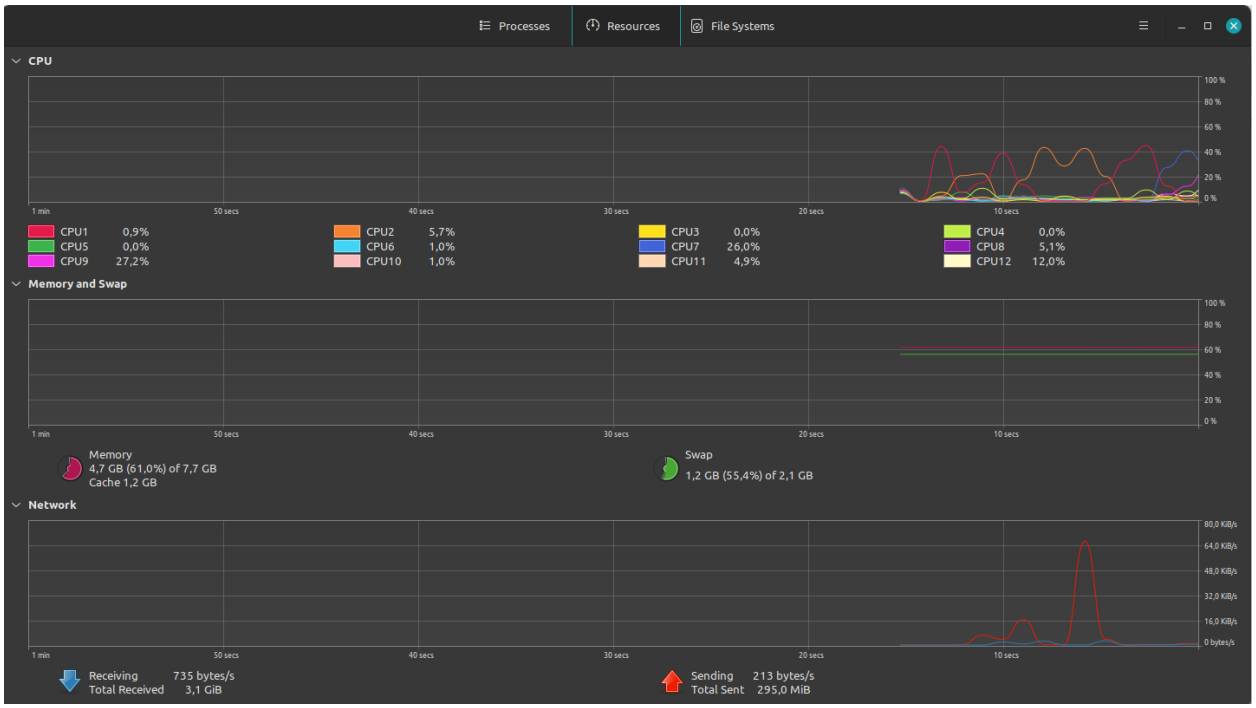
I make a mistake in here, I shouldn't copy log files from server to clients.






## Cpu check ( busy waiting):




Thanks to the inotify library, My program is not using lots of cpu. It means there is no busy waiting.

**Memory leak check with valgrind:**

```
gcc BibakBOXClient.o -o BibakBOXClient -pthread -lnotify
valgrind --leak-check=full ./BibakBOXClient [arguments]
==173883== Memcheck, a memory error detector
==173883== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==173883== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==173883== Command: ./BibakBOXClient [arguments]
==173883==
Usage: ./BibakBOXClient [clientDirPath] [portnumber] (optional)[server ip]
==173883==
==173883== HEAP SUMMARY:
==173883==     in use at exit: 0 bytes in 0 blocks
==173883==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==173883==
==173883== All heap blocks were freed -- no leaks are possible
==173883==
==173883== For lists of detected and suppressed errors, rerun with: -s
==173883== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
make: *** [makefile:19: check] Error 1
bulut@bulut:~/Desktop/repo/CSE344/final$
ph
```

The makefile I use for this: ( I added the “check” part).

```
FLAGS = -c -Wall -ansi -pedantic -errors -std=gnu99 -g
LIBS = -lnotify

all: BibakBOXClient BibakBOXServer

BibakBOXClient: BibakBOXClient.o
    gcc BibakBOXClient.o -o BibakBOXClient -pthread $(LIBS)

BibakBOXClient.o: BibakBOXClient.c
    gcc $(FLAGS) BibakBOXClient.c

BibakBOXServer: BibakBOXServer.o
    gcc BibakBOXServer.o -o BibakBOXServer -pthread

BibakBOXServer.o: BibakBOXServer.c
    gcc $(FLAGS) BibakBOXServer.c

check: BibakBOXClient
    valgrind --leak-check=full ./BibakBOXClient [arguments]

clean:
    rm -rf *.o BibakBOXClient BibakBOXServer
```

I couldn't test different devices because I did not  have time but it should work. I mean I suppose it will work..