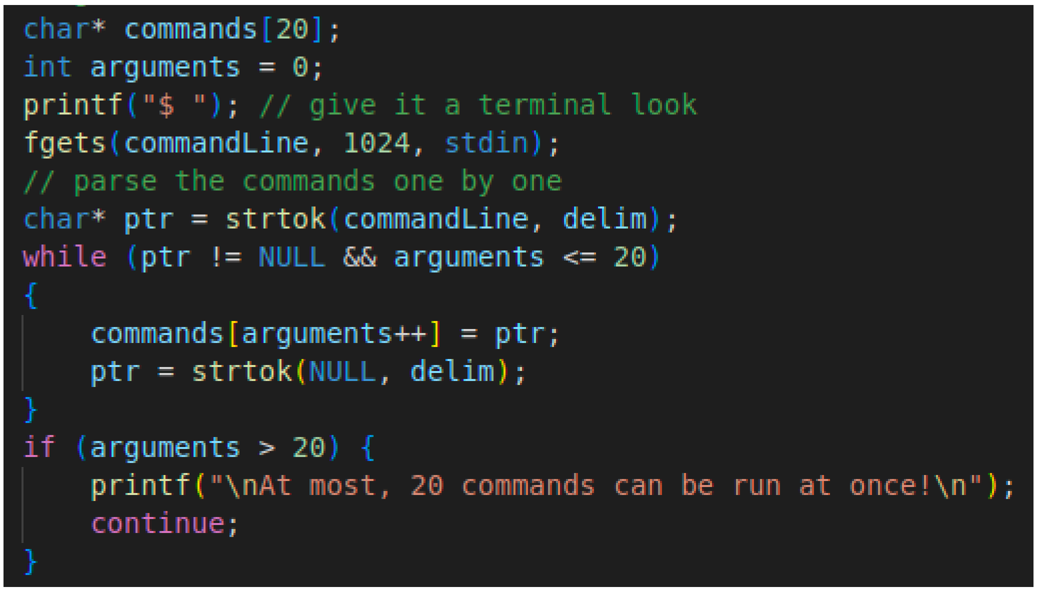
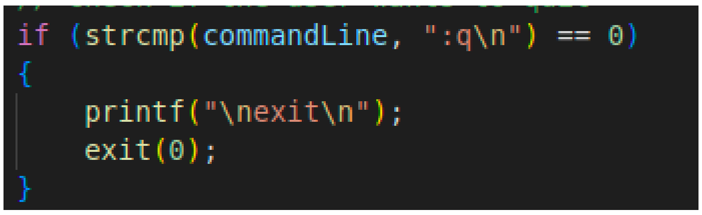
Code Explanation

I started by getting the command and parsing it, and if there are more than 20 commands, program asks for another set of commands.



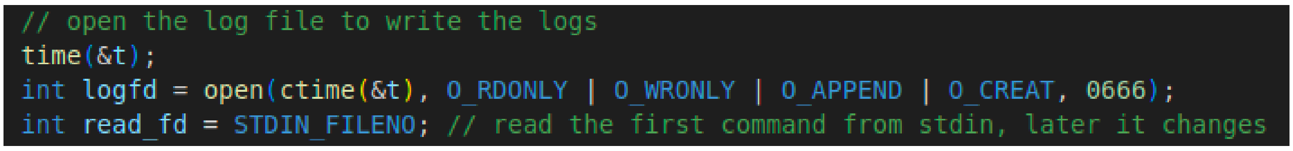
```c
char* commands[20];
int arguments = 0;
printf("$ "); // give it a terminal look
fgets(commandLine, 1024, stdin);
// parse the commands one by one
char* ptr = strtok(commandLine, delim);
while (ptr != NULL && arguments <= 20)
{
    commands[arguments++] = ptr;
    ptr = strtok(NULL, delim);
}
if (arguments > 20) {
    printf("\nAt most, 20 commands can be run at once!\n");
    continue;
}
```

If given command is :q then program terminates.



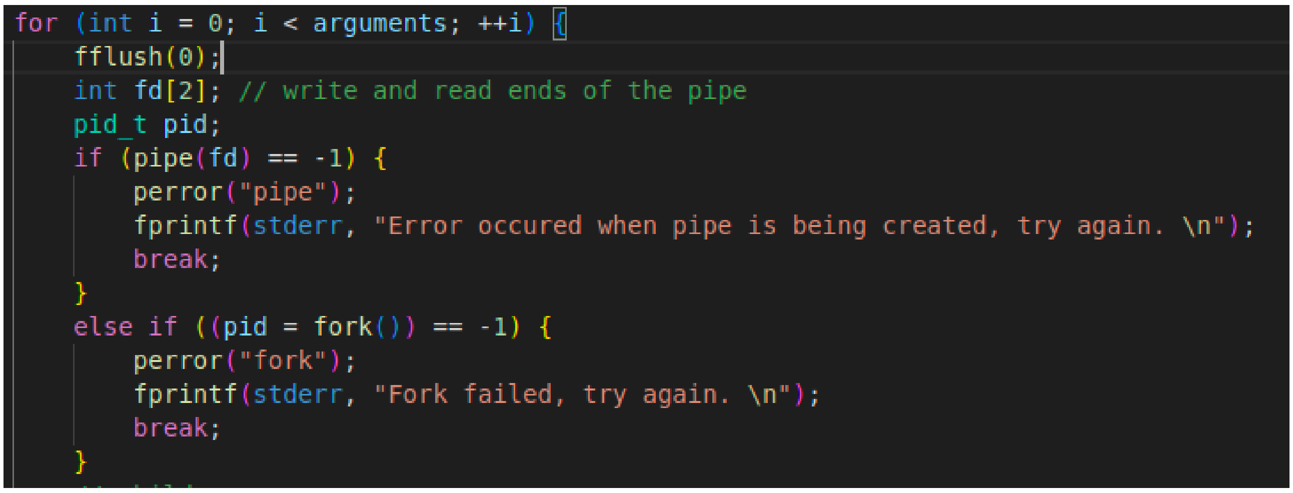
```c
if (strcmp(commandLine, ":q\n") == 0)
{
    printf("\nexit\n");
    exit(0);
}
```

Open logfile to write the executed commands and pid of the children.
read_fd is set to STDIN_FILENO at the beginning because child will read from stdin at the beginning, read_fd can be changed later via redirections (<).



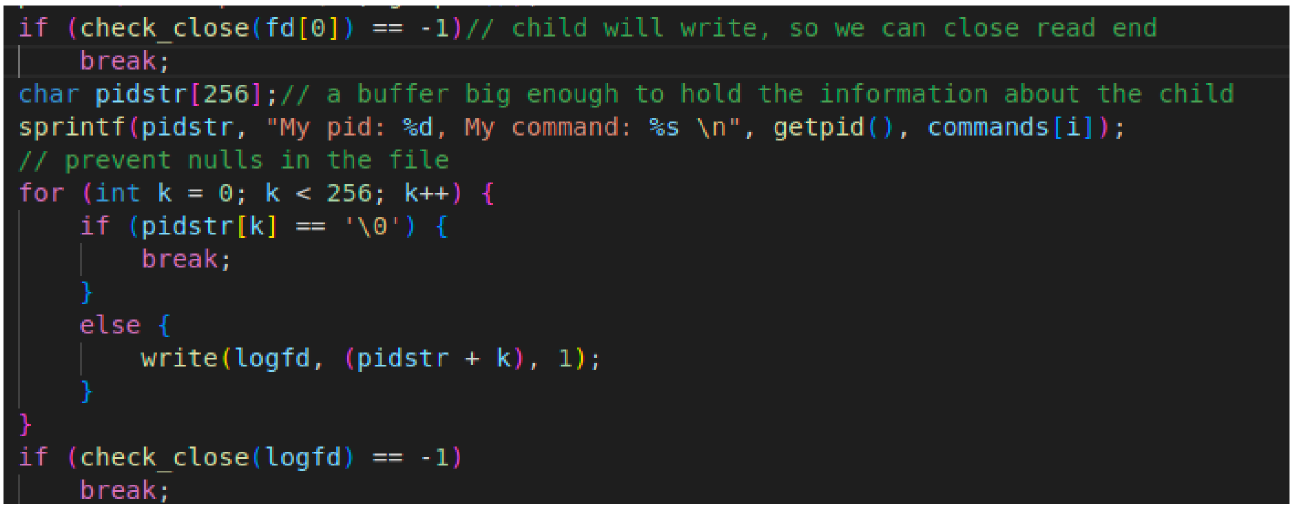
```c
// open the log file to write the logs
time(&t);
int logfd = open(ctime(&t), O_RDONLY | O_WRONLY | O_APPEND | O_CREAT, 0666);
int read_fd = STDIN_FILENO; // read the first command from stdin, later it changes
```

After that each command is run one by one in a for loop by creating a new child for a command. Firstly error checks:



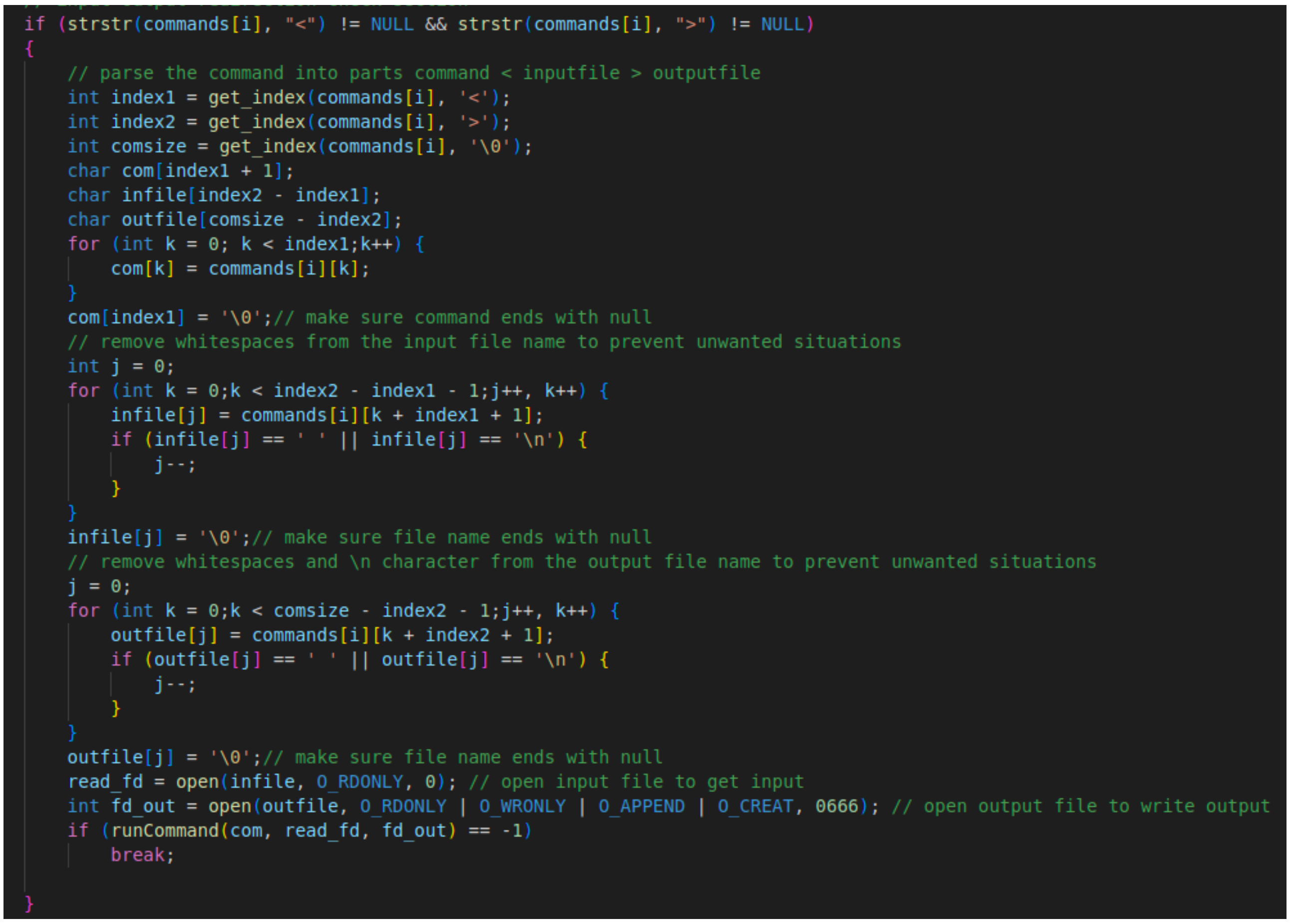
```c
for (int i = 0; i < arguments; ++i) {
    fflush(0);
    int fd[2]; // write and read ends of the pipe
    pid_t pid;
    if (pipe(fd) == -1) {
        perror("pipe");
        fprintf(stderr, "Error occured when pipe is being created, try again. \n");
        break;
    }
    else if ((pid = fork()) == -1) {
        perror("fork");
        fprintf(stderr, "Fork failed, try again. \n");
        break;
    }
```

If pipe is created successfully and child is created then command can be executed in child process.
Here child writes its pid and command to the log file. And closes the file.



```c
if (check_close(fd[0]) == -1)// child will write, so we can close read end
    break;
char pidstr[256];// a buffer big enough to hold the information about the child
sprintf(pidstr, "My pid: %d, My command: %s \n", getpid(), commands[i]);
// prevent nulls in the file
for (int k = 0; k < 256; k++) {
    if (pidstr[k] == '\0') {
        break;
    }
    else {
        write(logfd, (pidstr + k), 1);
    }
}
if (check_close(logfd) == -1)
    break;
```

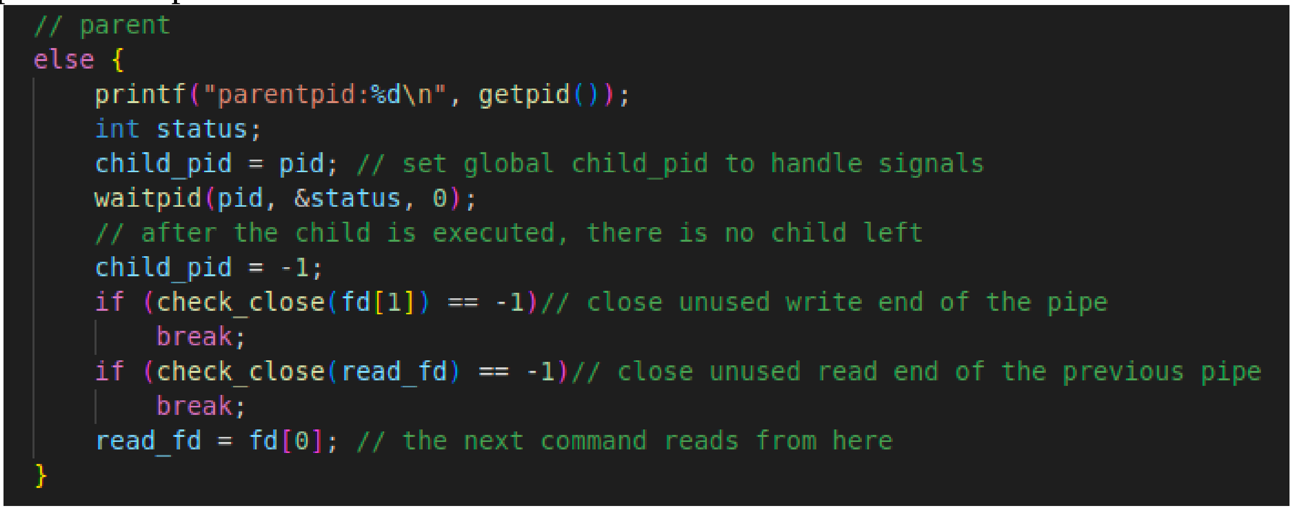
And this is how directions are handled. Firstly get the indices. This is for parsing command, input file and output file if a command like cat < in.txt > out.txt is given. Com will be “cat”, infile will be “in.txt” outfile will be “out.txt” and read_fd will be changed to file descriptor that opened in.txt .



```c
if (strstr(commands[i], "<") != NULL && strstr(commands[i], ">") != NULL)
{
    // parse the command into parts command < inputfile > outputfile
    int index1 = get_index(commands[i], '<');
    int index2 = get_index(commands[i], '>');
    int comsize = get_index(commands[i], '\0');
    char com[index1 + 1];
    char infile[index2 - index1];
    char outfile[comsize - index2];
    for (int k = 0; k < index1;k++) {
        com[k] = commands[i][k];
    }
    com[index1] = '\0';// make sure command ends with null
    // remove whitespaces from the input file name to prevent unwanted situations
    int j = 0;
    for (int k = 0;k < index2 - index1 - 1;j++, k++) {
        infile[j] = commands[i][k + index1 + 1];
        if (infile[j] == ' ' || infile[j] == '\n') {
            j--;
        }
    }
    infile[j] = '\0';// make sure file name ends with null
    // remove whitespaces and \n character from the output file name to prevent unwanted situations
    j = 0;
    for (int k = 0;k < comsize - index2 - 1;j++, k++) {
        outfile[j] = commands[i][k + index2 + 1];
        if (outfile[j] == ' ' || outfile[j] == '\n') {
            j--;
        }
    }
    outfile[j] = '\0';// make sure file name ends with null
    read_fd = open(infile, O_RDONLY, 0); // open input file to get input
    int fd_out = open(outfile, O_RDONLY | O_WRONLY | O_APPEND | O_CREAT, 0666); // open output file to write output
    if (runCommand(com, read_fd, fd_out) == -1)
        break;

}
```

If command includes only one of < or > then similar things happen so that I don’t put the screenshot but the code handles it and I commented on them as well to prevent confusion.

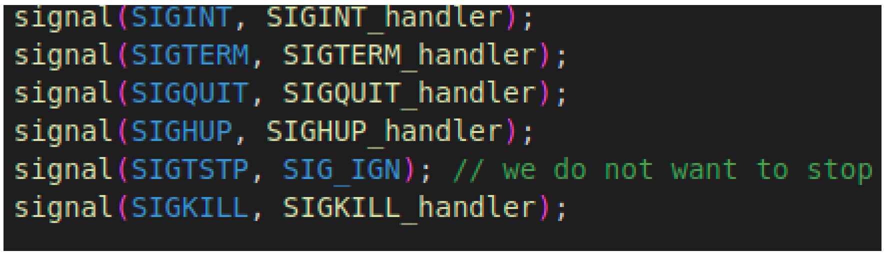
In parent process firstly wait for the child and then close the unused file descriptors after that change read_fd to give the last executed process’s output to the next process’s input.



```c
// parent
else {
    printf("parentpid:%d\n", getpid());
    int status;
    child_pid = pid; // set global child_pid to handle signals
    waitpid(pid, &status, 0);
    // after the child is executed, there is no child left
    child_pid = -1;
    if (check_close(fd[1]) == -1)// close unused write end of the pipe
        break;
    if (check_close(read_fd) == -1)// close unused read end of the previous pipe
        break;
    read_fd = fd[0]; // the next command reads from here
}
```

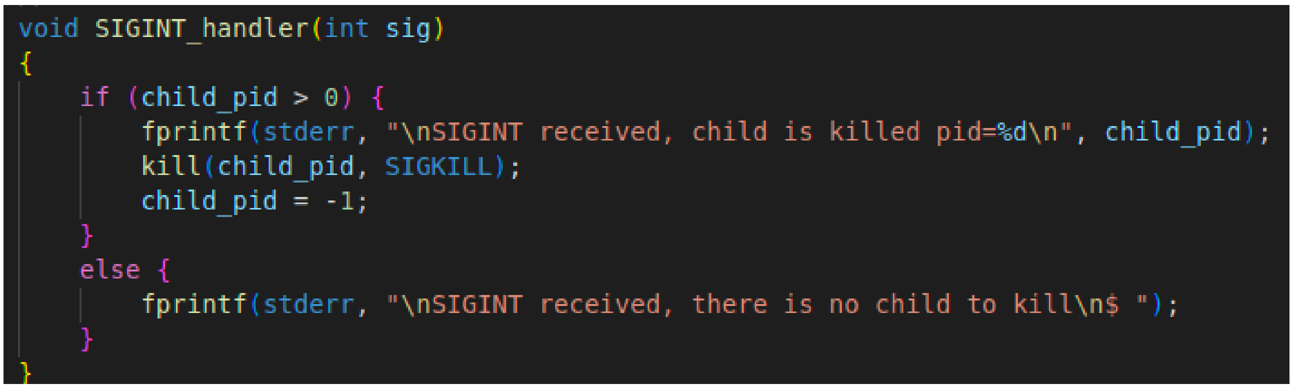
Signals are handled as below

At the beginning of the program, handler for each signal is specified



```c
signal(SIGINT, SIGINT_handler);
signal(SIGTERM, SIGTERM_handler);
signal(SIGQUIT, SIGQUIT_handler);
signal(SIGHUP, SIGHUP_handler);
signal(SIGTSTP, SIG_IGN); // we do not want to stop
signal(SIGKILL, SIGKILL_handler);
```

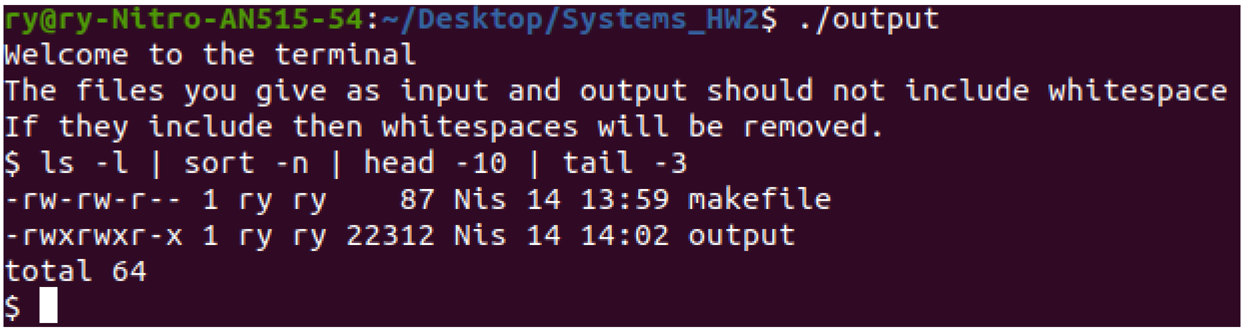
Handlers are implemented as below, all of them has the same structure, only the messages are different specifying which signal is received.
Also there is a handler for SIGKILL but this will not affect anything because SIGKILL cannot be handled or blocked.



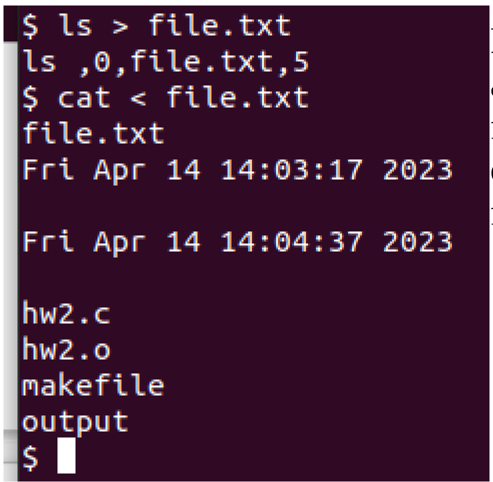
```c
void SIGINT_handler(int sig)
{
    if (child_pid > 0) {
        fprintf(stderr, "\nSIGINT received, child is killed pid=%d\n", child_pid);
        kill(child_pid, SIGKILL);
        child_pid = -1;
    }
    else {
        fprintf(stderr, "\nSIGINT received, there is no child to kill\n$ ");
    }
}
```

TESTS

When 4 commands are given using pipes, program works very well.

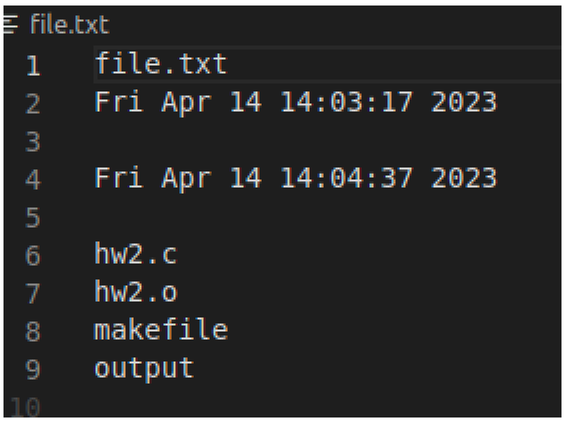


```
ry@ry-Nitro-AN515-54:~/Desktop/Systems_HW2$ ./output
Welcome to the terminal
The files you give as input and output should not include whitespace
If they include then whitespaces will be removed.
$ ls -l | sort -n | head -10 | tail -3
-rw-rw-r-- 1 ry ry    87 Nis 14 13:59 makefile
-rwxrwxr-x 1 ry ry 22312 Nis 14 14:02 output
total 64
$ 
```



```
$ ls > file.txt
ls ,0,file.txt,5
$ cat < file.txt
file.txt
Fri Apr 14 14:03:17 2023

Fri Apr 14 14:04:37 2023

hw2.c
hw2.o
makefile
output
$ 
```

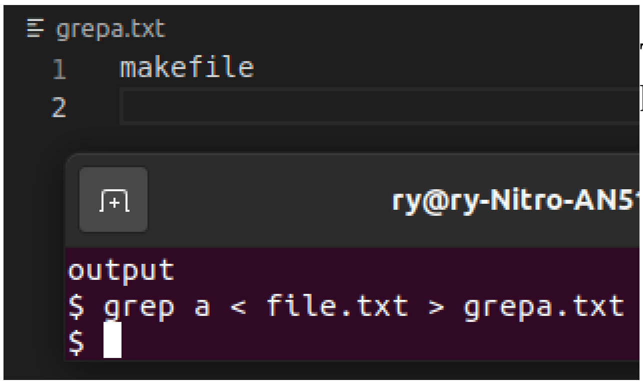
Redirections work as well: after ls > file.txt command file.txt is created and output of ls is written to file.txt therefore,



```
file.txt
Fri Apr 14 14:03:17 2023

Fri Apr 14 14:04:37 2023

hw2.c
hw2.o
makefile
output
```

cat < file.txt writes the same thing that ls would write to the terminal

What happens if both of the redirections are used:

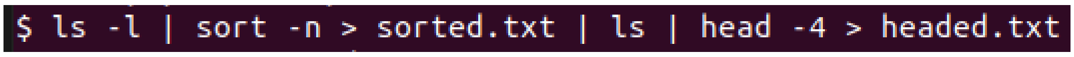


```
grepa.txt
makefile

```

```
ry@ry-Nitro-AN5
output
$ grep a < file.txt > grepa.txt
$ 
```

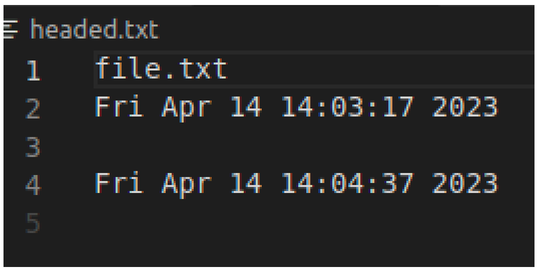
This works as well, it writes the words that has a in file.txt to grepa.txt successfully.

If a command that involves redirections and pipes at the same time is given



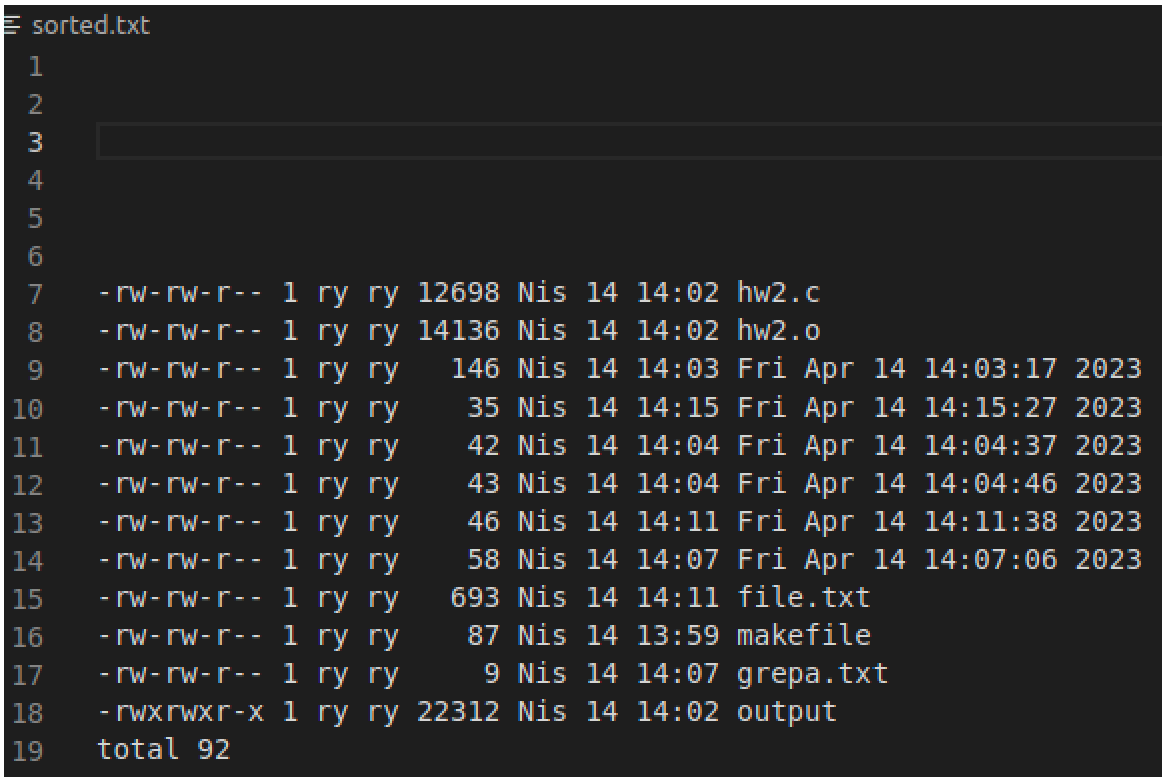
```
$ ls -l | sort -n > sorted.txt | ls | head -4 > headed.txt
```

All of them are handled



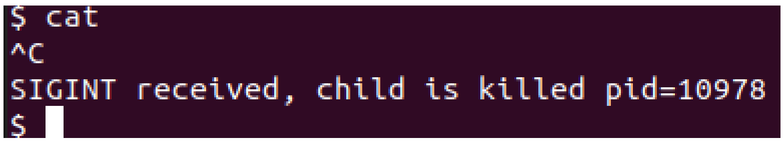
```
headed.txt
file.txt
Fri Apr 14 14:03:17 2023

Fri Apr 14 14:04:37 2023

```

There are spaces at the beginning but this is rhe same result when ls -l is written to linux terminal.



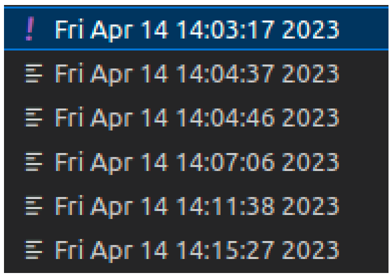
```
sorted.txt






-rw-rw-r-- 1 ry ry 12698 Nis 14 14:02 hw2.c
-rw-rw-r-- 1 ry ry 14136 Nis 14 14:02 hw2.o
-rw-rw-r-- 1 ry ry   146 Nis 14 14:03 Fri Apr 14 14:03:17 2023
-rw-rw-r-- 1 ry ry    35 Nis 14 14:15 Fri Apr 14 14:15:27 2023
-rw-rw-r-- 1 ry ry    42 Nis 14 14:04 Fri Apr 14 14:04:37 2023
-rw-rw-r-- 1 ry ry    43 Nis 14 14:04 Fri Apr 14 14:04:46 2023
-rw-rw-r-- 1 ry ry    46 Nis 14 14:11 Fri Apr 14 14:11:38 2023
-rw-rw-r-- 1 ry ry    58 Nis 14 14:07 Fri Apr 14 14:07:06 2023
-rw-rw-r-- 1 ry ry   693 Nis 14 14:11 file.txt
-rw-rw-r-- 1 ry ry    87 Nis 14 13:59 makefile
-rw-rw-r-- 1 ry ry     9 Nis 14 14:07 grepa.txt
-rwxrwxr-x 1 ry ry 22312 Nis 14 14:02 output
total 92
```

When cat is run without any input then program stuck in child proccess so that means signals can be tested here. If ctrl + c is pressed, then child is killed and program is available to get commands again. This is valid for all signals except SIGSTP, SIGSTP is ignored.



```
$ cat
^C
SIGINT received, child is killed pid=10978
$ 
```

Also, log files are created at the end of execution of each line



```
! Fri Apr 14 14:03:17 2023
Fri Apr 14 14:04:37 2023
Fri Apr 14 14:04:46 2023
Fri Apr 14 14:07:06 2023
Fri Apr 14 14:11:38 2023
Fri Apr 14 14:15:27 2023
```