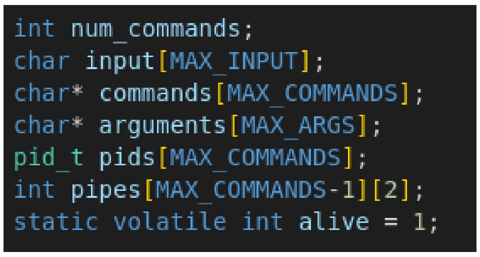
CSE 344
HOMEWORK2
REPORT

Ömer Faruk SAYAR
171044038

Flow of the program

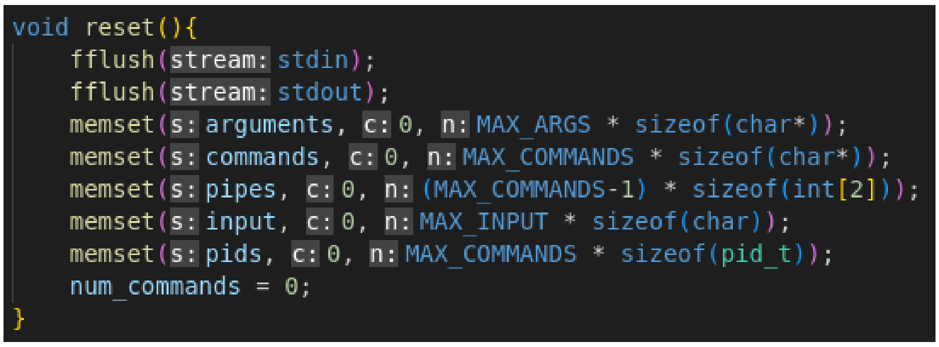
In a loop:

Get Input (Parent) => Parse Commands (Parent) => Create Child Processes (Parent) => Connect pipes to stdin and stdout (Child) => Check and set redirection from stdin/stdout to file (Child) => Execv bin/sh command given from user (Child) => Wait until all child processes to finish (Parent) => Write the log file => Cleanup for the next command.



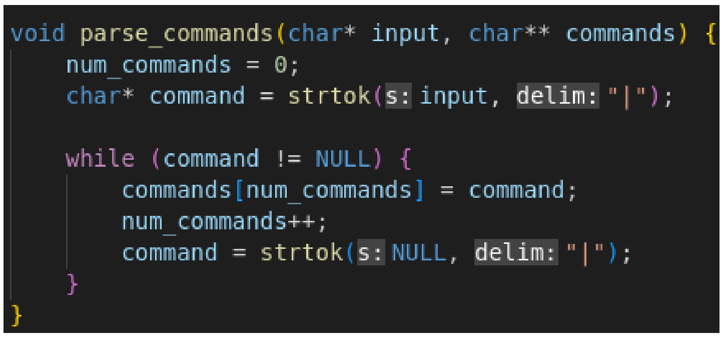
```c
int num_commands;
char input[MAX_INPUT];
char* commands[MAX_COMMANDS];
char* arguments[MAX_ARGS];
pid_t pids[MAX_COMMANDS];
int pipes[MAX_COMMANDS-1][2];
static volatile int alive = 1;
```

There are 6 global variable to handle input, commands, pipes and pids. After getting an input from the user, program resets them using reset function.



```c
void reset(){
    fflush(stream: stdin);
    fflush(stream: stdout);
    memset(s: arguments, c: 0, n: MAX_ARGS * sizeof(char*));
    memset(s: commands, c: 0, n: MAX_COMMANDS * sizeof(char*));
    memset(s: pipes, c: 0, n: (MAX_COMMANDS-1) * sizeof(int[2]));
    memset(s: input, c: 0, n: MAX_INPUT * sizeof(char));
    memset(s: pids, c: 0, n: MAX_COMMANDS * sizeof(pid_t));
    num_commands = 0;
}
```

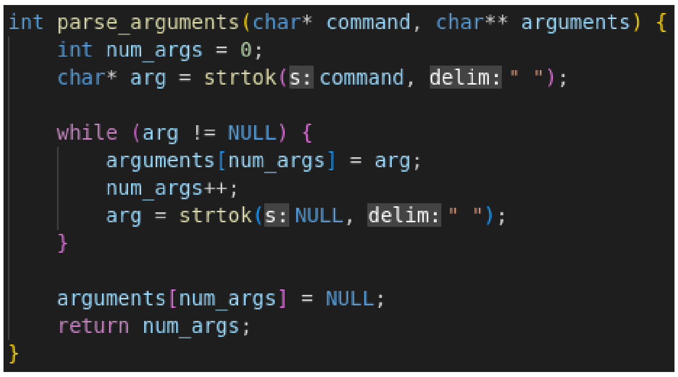
After an input getting from the user program checks if it is a proper input or not. Also checks if it is a quit command. Then calls pars_commands function to separate commands according to pipe (|) operator.



```c
void parse_commands(char* input, char** commands) {
    num_commands = 0;
    char* command = strtok(s: input, delim: "|");

    while (command != NULL) {
        commands[num_commands] = command;
        num_commands++;
        command = strtok(s: NULL, delim: "|");
    }
}
```

Then creates pipes according to the number of command.

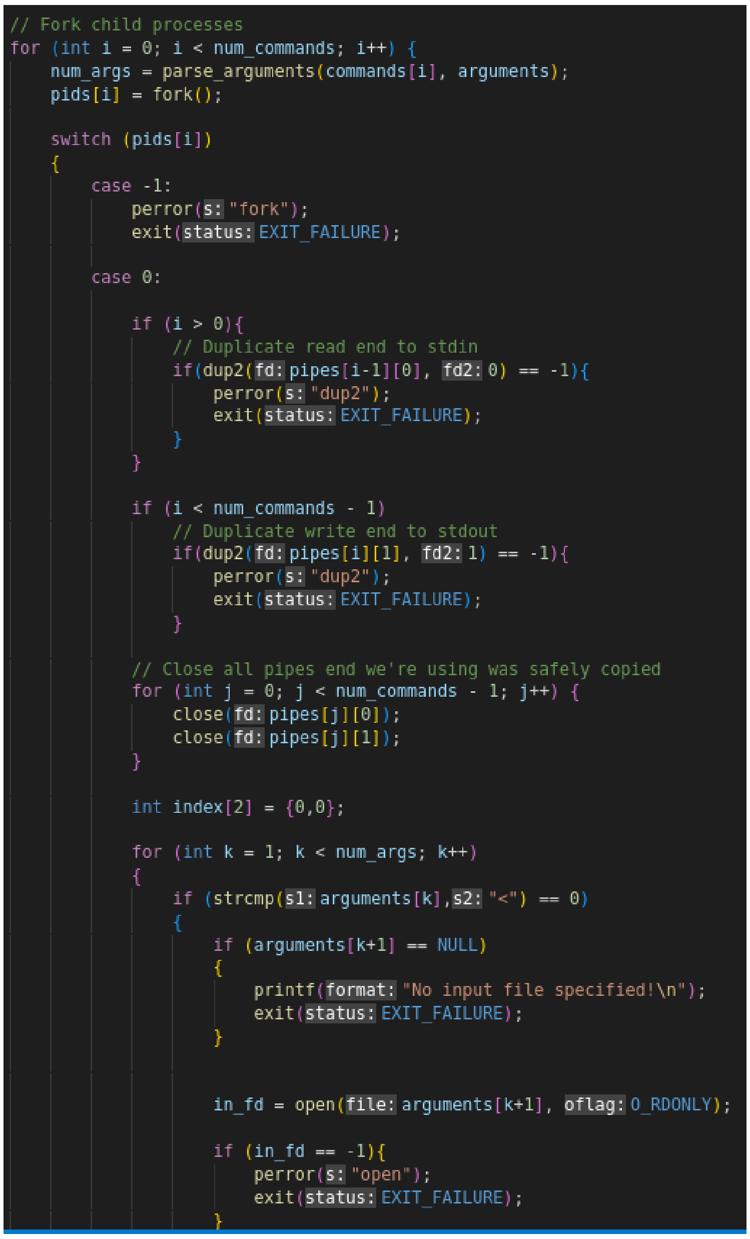
After that, in a for loop that runs number of command times, program parse the arguments of a command according to whitespace (“ ”) .



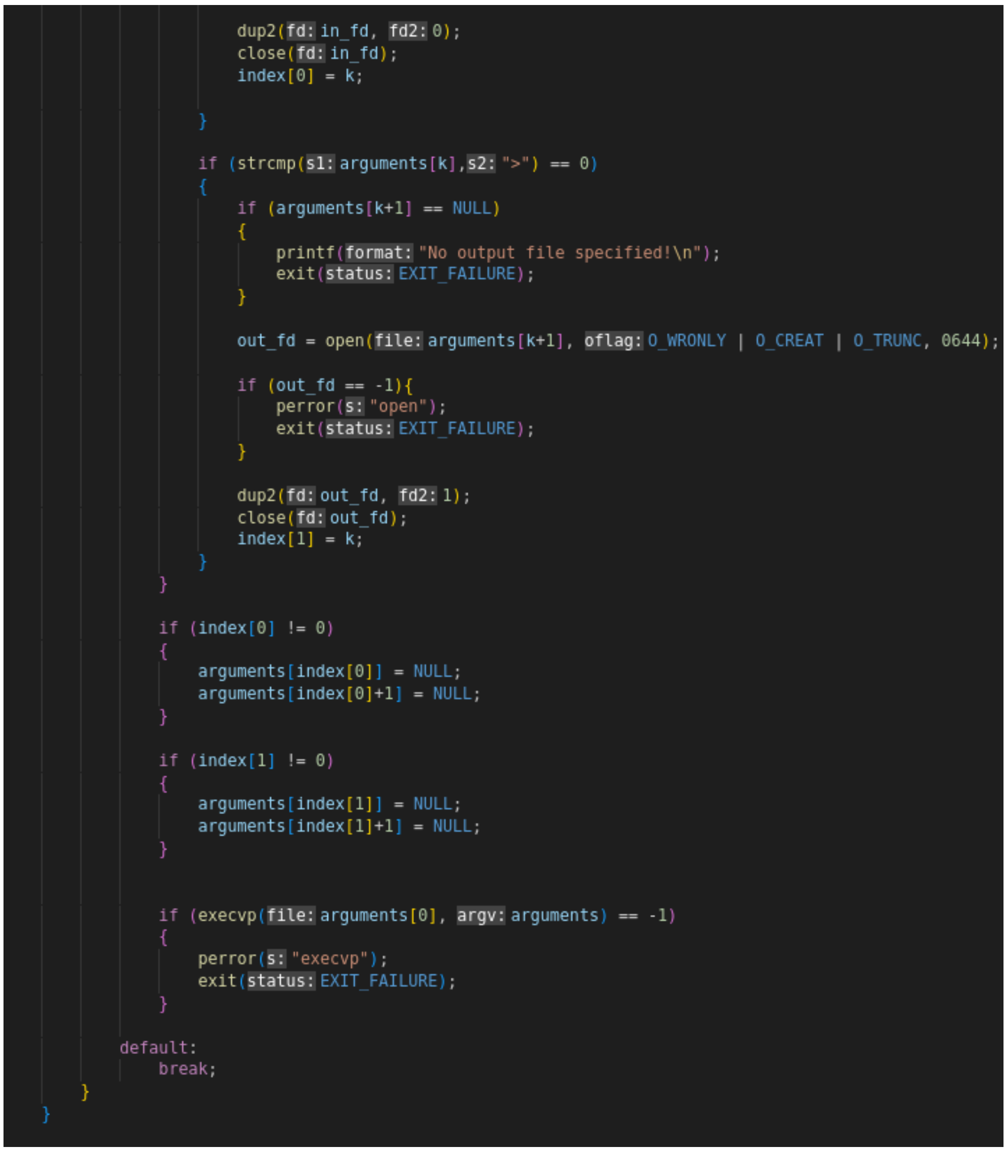
```
int parse_arguments(char* command, char** arguments) {
    int num_args = 0;
    char* arg = strtok(s: command, delim: " ");

    while (arg != NULL) {
        arguments[num_args] = arg;
        num_args++;
        arg = strtok(s: NULL, delim: " ");
    }

    arguments[num_args] = NULL;
    return num_args;
}
```

After arguments are set, program creates a child process using fork().
In the child process, connects stdin to read end of the pipe and stdout to write end of the pipe using dup2() and closes pipe file descriptors.

Then checks if any redirection operator in the arguments and if there is any, program handles properly by redirecting stdin or stdout to appropriate file descriptor using dup2 again.



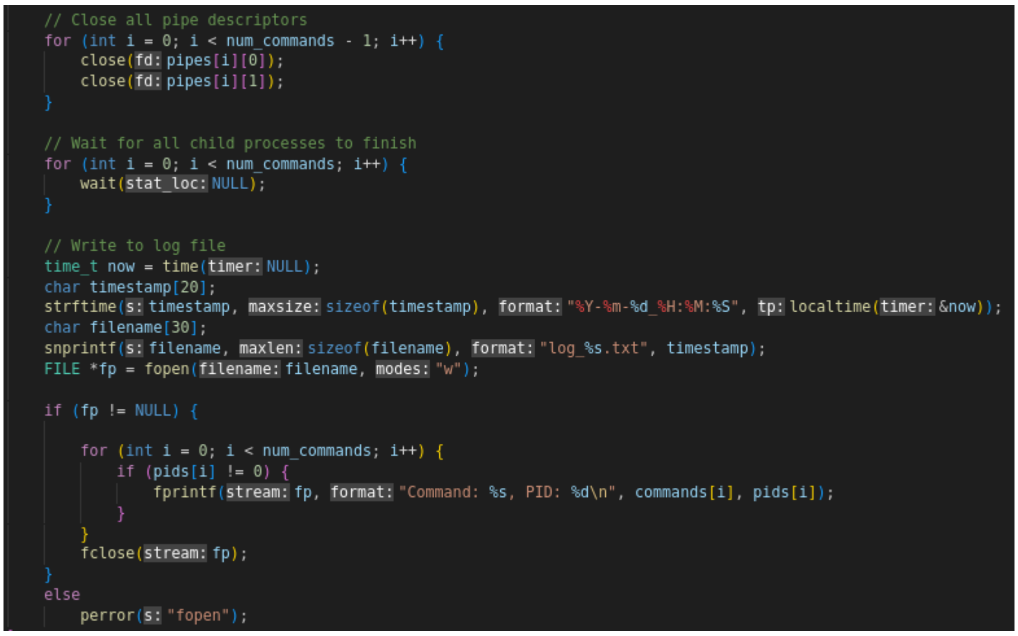
```
// Fork child processes
for (int i = 0; i < num_commands; i++) {
    num_args = parse_arguments(commands[i], arguments);
    pids[i] = fork();

    switch (pids[i])
    {
        case -1:
            perror(s: "fork");
            exit(status: EXIT_FAILURE);

        case 0:

            if (i > 0){
                // Duplicate read end to stdin
                if(dup2(fd: pipes[i-1][0], fd2: 0) == -1){
                    perror(s: "dup2");
                    exit(status: EXIT_FAILURE);
                }
            }

            if (i < num_commands - 1)
                // Duplicate write end to stdout
                if(dup2(fd: pipes[i][1], fd2: 1) == -1){
                    perror(s: "dup2");
                    exit(status: EXIT_FAILURE);
                }

            // Close all pipes end we're using was safely copied
            for (int j = 0; j < num_commands - 1; j++) {
                close(fd: pipes[j][0]);
                close(fd: pipes[j][1]);
            }

            int index[2] = {0,0};

            for (int k = 1; k < num_args; k++)
            {
                if (strcmp(s1: arguments[k],s2: "<") == 0)
                {
                    if (arguments[k+1] == NULL)
                    {
                        printf(format: "No input file specified!\n");
                        exit(status: EXIT_FAILURE);
                    }


                    in_fd = open(file: arguments[k+1], oflag: O_RDONLY);

                    if (in_fd == -1){
                        perror(s: "open");
                        exit(status: EXIT_FAILURE);
                    }
```



```
                    dup2(fd: in_fd, fd2: 0);
                    close(fd: in_fd);
                    index[0] = k;

                }

                if (strcmp(s1: arguments[k],s2: ">") == 0)
                {
                    if (arguments[k+1] == NULL)
                    {
                        printf(format: "No output file specified!\n");
                        exit(status: EXIT_FAILURE);
                    }

                    out_fd = open(file: arguments[k+1], oflag: O_WRONLY | O_CREAT | O_TRUNC, 0644);

                    if (out_fd == -1){
                        perror(s: "open");
                        exit(status: EXIT_FAILURE);
                    }

                    dup2(fd: out_fd, fd2: 1);
                    close(fd: out_fd);
                    index[1] = k;
                }
            }

            if (index[0] != 0)
            {
                arguments[index[0]] = NULL;
                arguments[index[0]+1] = NULL;
            }

            if (index[1] != 0)
            {
                arguments[index[1]] = NULL;
                arguments[index[1]+1] = NULL;
            }


            if (execvp(file: arguments[0], argv: arguments) == -1)
            {
                perror(s: "execvp");
                exit(status: EXIT_FAILURE);
            }

        default:
            break;
    }
}
```

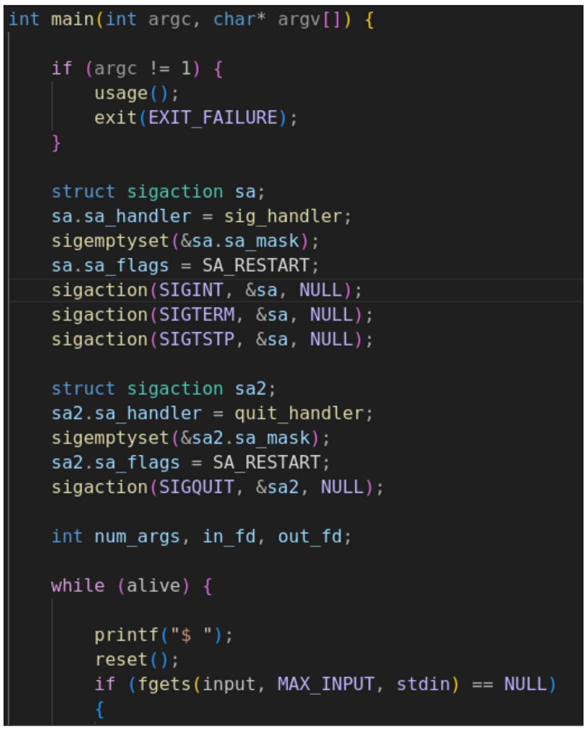
After the pipe and redirection operators handled, program calls execvp (I did not use execl because it was needed to use execl with bin/sh command and it creates a shell process then creates a child process of the shell to execute the command, Instractor gave permission to use execvp) to run given bin/sh command with the arguments.

Meanwhile, after closing it's pipe fds, parent process wait for all child process to finish then pids of child processes with their corresponding commands logged in a file named current timestamp.

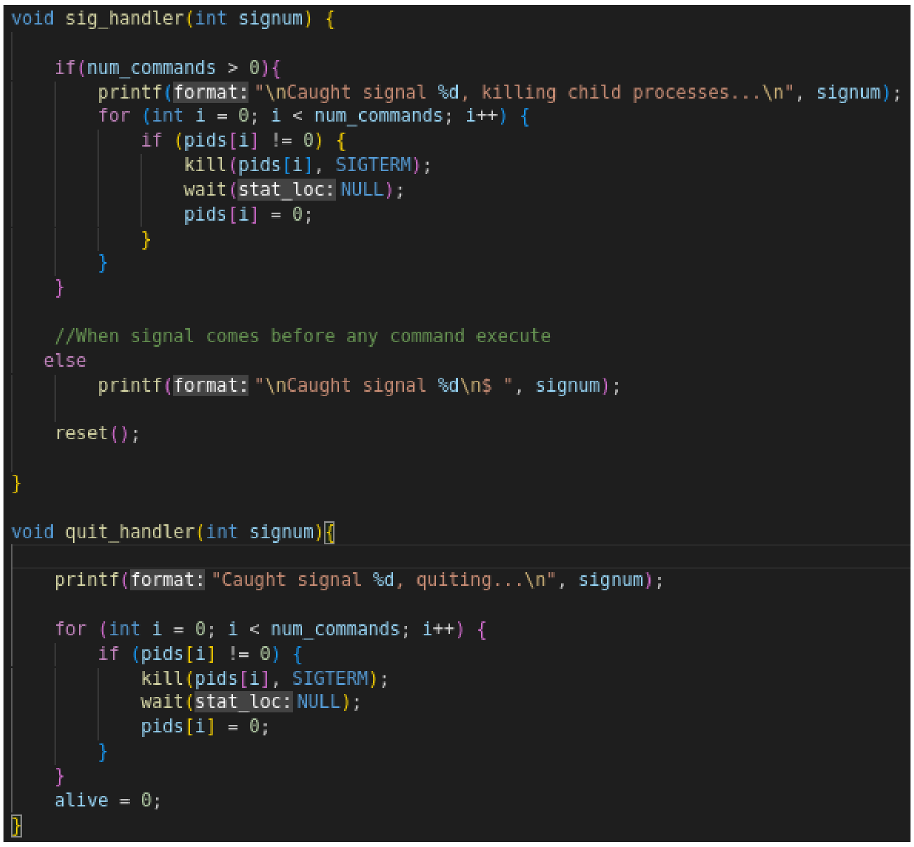


```
// Close all pipe descriptors
for (int i = 0; i < num_commands - 1; i++) {
    close(fd: pipes[i][0]);
    close(fd: pipes[i][1]);
}

// Wait for all child processes to finish
for (int i = 0; i < num_commands; i++) {
    wait(stat_loc: NULL);
}

// Write to log file
time_t now = time(timer: NULL);
char timestamp[20];
strftime(s: timestamp, maxsize: sizeof(timestamp), format: "%Y-%m-%d_%H:%M:%S", tp: localtime(timer: &now));
char filename[30];
snprintf(s: filename, maxlen: sizeof(filename), format: "log_%s.txt", timestamp);
FILE *fp = fopen(filename: filename, modes: "w");

if (fp != NULL) {

    for (int i = 0; i < num_commands; i++) {
        if (pids[i] != 0) {
            fprintf(stream: fp, format: "Command: %s, PID: %d\n", commands[i], pids[i]);
        }
    }
    fclose(stream: fp);
}
else
    perror(s: "fopen");
```

Program have also two signal handler for four different signals. When a SIGTERM or SIGTSTP or SIGINT signal comes, handler kills all child then program return to prompt. When a SIGQUIT signal comes, program kills all child then terminates the program properly.

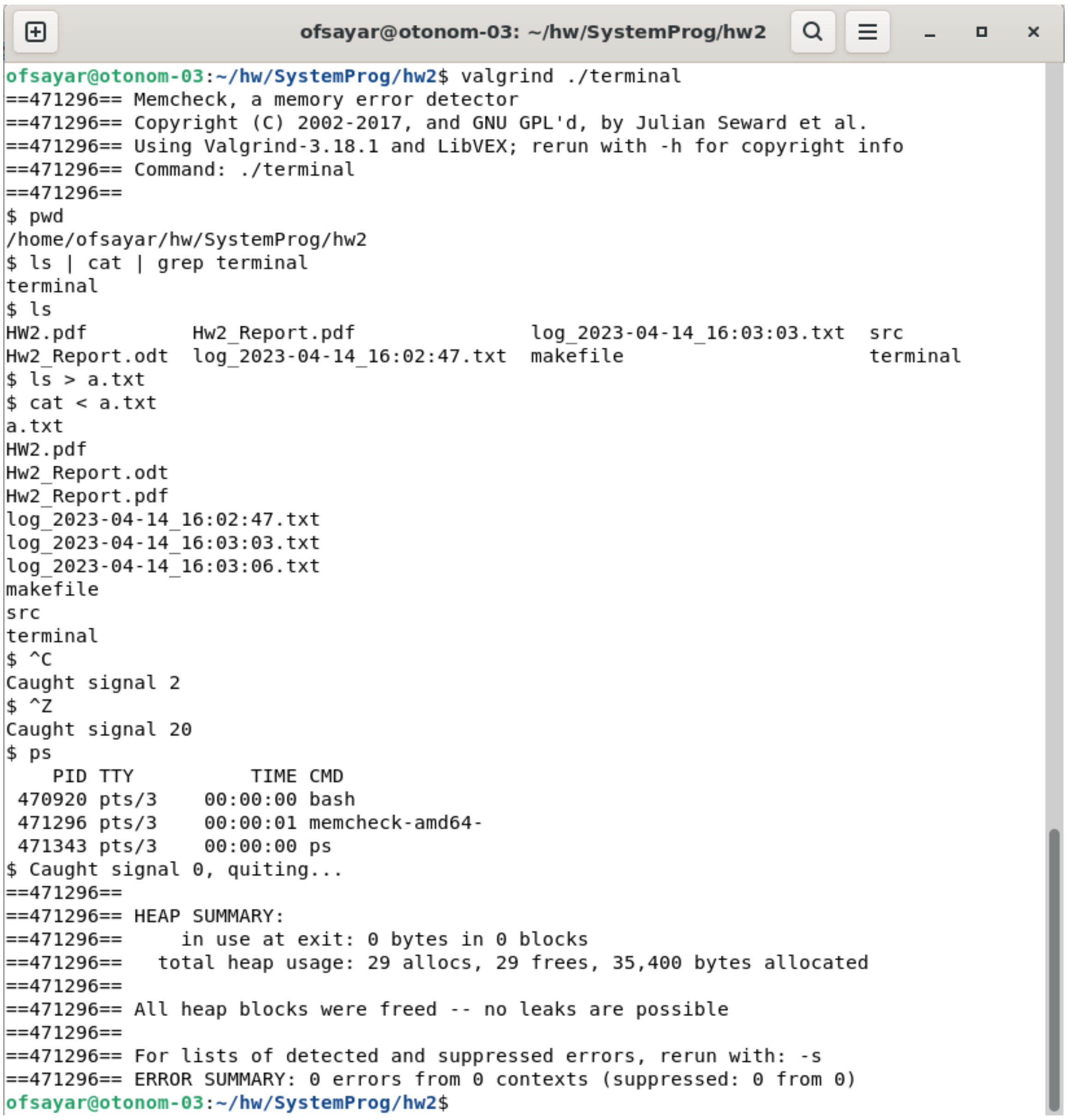


```c
int main(int argc, char* argv[]) {

    if (argc != 1) {
        usage();
        exit(EXIT_FAILURE);
    }

    struct sigaction sa;
    sa.sa_handler = sig_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sigaction(SIGINT, &sa, NULL);
    sigaction(SIGTERM, &sa, NULL);
    sigaction(SIGTSTP, &sa, NULL);

    struct sigaction sa2;
    sa2.sa_handler = quit_handler;
    sigemptyset(&sa2.sa_mask);
    sa2.sa_flags = SA_RESTART;
    sigaction(SIGQUIT, &sa2, NULL);

    int num_args, in_fd, out_fd;

    while (alive) {

        printf("$ ");
        reset();
        if (fgets(input, MAX_INPUT, stdin) == NULL)
        {
```



```c
void sig_handler(int signum) {

    if(num_commands > 0){
        printf(format: "\nCaught signal %d, killing child processes...\n", signum);
        for (int i = 0; i < num_commands; i++) {
            if (pids[i] != 0) {
                kill(pids[i], SIGTERM);
                wait(stat_loc: NULL);
                pids[i] = 0;
            }
        }
    }

    //When signal comes before any command execute
   else
        printf(format: "\nCaught signal %d\n$ ", signum);

    reset();

}

void quit_handler(int signum){

    printf(format: "Caught signal %d, quiting...\n", signum);

    for (int i = 0; i < num_commands; i++) {
        if (pids[i] != 0) {
            kill(pids[i], SIGTERM);
            wait(stat_loc: NULL);
            pids[i] = 0;
        }
    }
    alive = 0;
}
```

Valgrind Output :

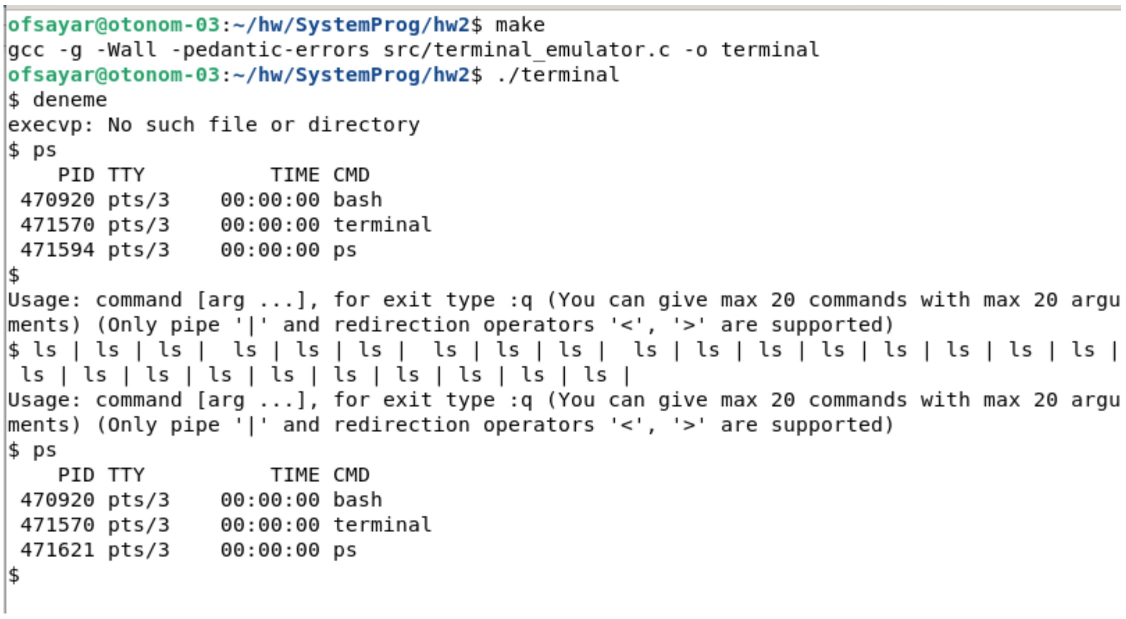


ofsayar@otonom-03: ~/hw/SystemProg/hw2

```
ofsayar@otonom-03:~/hw/SystemProg/hw2$ valgrind ./terminal
==471296== Memcheck, a memory error detector
==471296== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==471296== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==471296== Command: ./terminal
==471296==
$ pwd
/home/ofsayar/hw/SystemProg/hw2
$ ls | cat | grep terminal
terminal
$ ls
HW2.pdf         Hw2_Report.pdf              log_2023-04-14_16:03:03.txt  src
Hw2_Report.odt  log_2023-04-14_16:02:47.txt  makefile                     terminal
$ ls > a.txt
$ cat < a.txt
a.txt
HW2.pdf
Hw2_Report.odt
Hw2_Report.pdf
log_2023-04-14_16:02:47.txt
log_2023-04-14_16:03:03.txt
log_2023-04-14_16:03:06.txt
makefile
src
terminal
$ ^C
Caught signal 2
$ ^Z
Caught signal 20
$ ps
    PID TTY          TIME CMD
 470920 pts/3    00:00:00 bash
 471296 pts/3    00:00:01 memcheck-amd64-
 471343 pts/3    00:00:00 ps
$ Caught signal 0, quiting...
==471296==
==471296== HEAP SUMMARY:
==471296==     in use at exit: 0 bytes in 0 blocks
==471296==   total heap usage: 29 allocs, 29 frees, 35,400 bytes allocated
==471296==
==471296== All heap blocks were freed -- no leaks are possible
==471296==
==471296== For lists of detected and suppressed errors, rerun with: -s
==471296== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
ofsayar@otonom-03:~/hw/SystemProg/hw2$
```

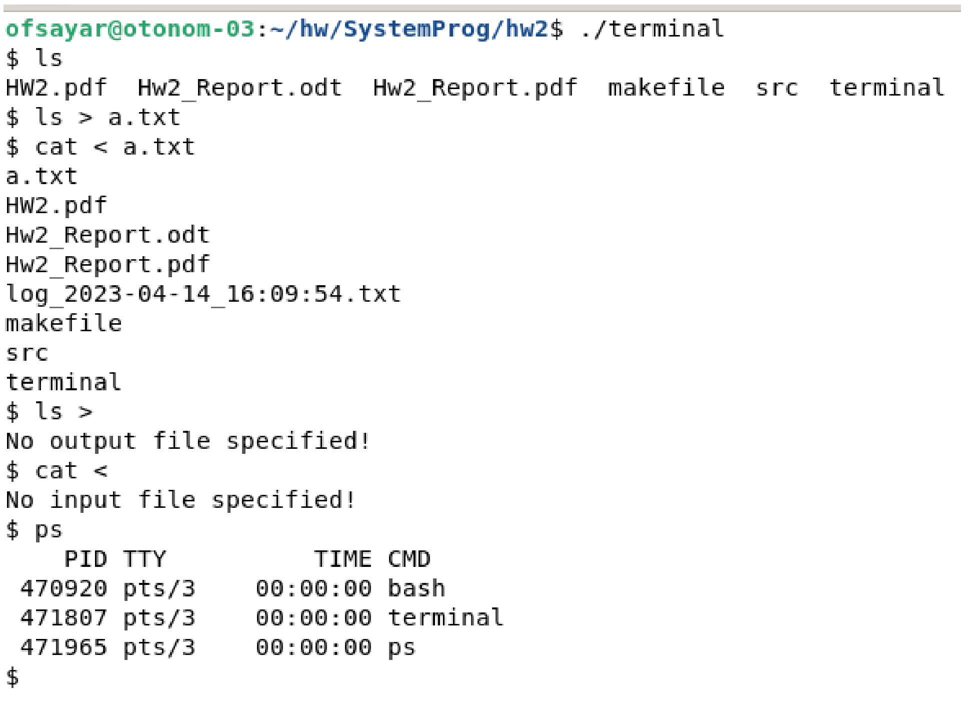
Test Cases:

1) Giving a non existing command – empty command – more than 20 command:



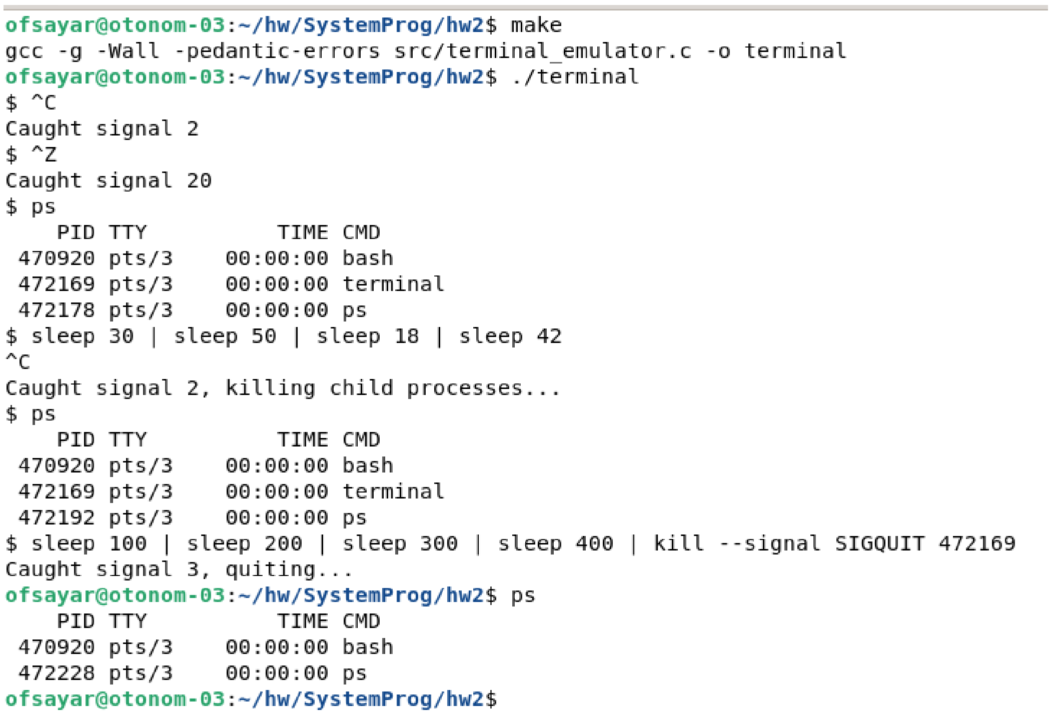
```
ofsayar@otonom-03:~/hw/SystemProg/hw2$ make
gcc -g -Wall -pedantic-errors src/terminal_emulator.c -o terminal
ofsayar@otonom-03:~/hw/SystemProg/hw2$ ./terminal
$ deneme
execvp: No such file or directory
$ ps
    PID TTY          TIME CMD
 470920 pts/3    00:00:00 bash
 471570 pts/3    00:00:00 terminal
 471594 pts/3    00:00:00 ps
$
Usage: command [arg ...], for exit type :q (You can give max 20 commands with max 20 argu
ments) (Only pipe '|' and redirection operators '<', '>' are supported)
$ ls | ls | ls |  ls | ls | ls |  ls | ls | ls |  ls | ls | ls | ls | ls | ls | ls | ls |
 ls | ls | ls | ls | ls | ls | ls | ls | ls | ls |
Usage: command [arg ...], for exit type :q (You can give max 20 commands with max 20 argu
ments) (Only pipe '|' and redirection operators '<', '>' are supported)
$ ps
    PID TTY          TIME CMD
 470920 pts/3    00:00:00 bash
 471570 pts/3    00:00:00 terminal
 471621 pts/3    00:00:00 ps
$
```

2) Redirection - Redirection to nowhere



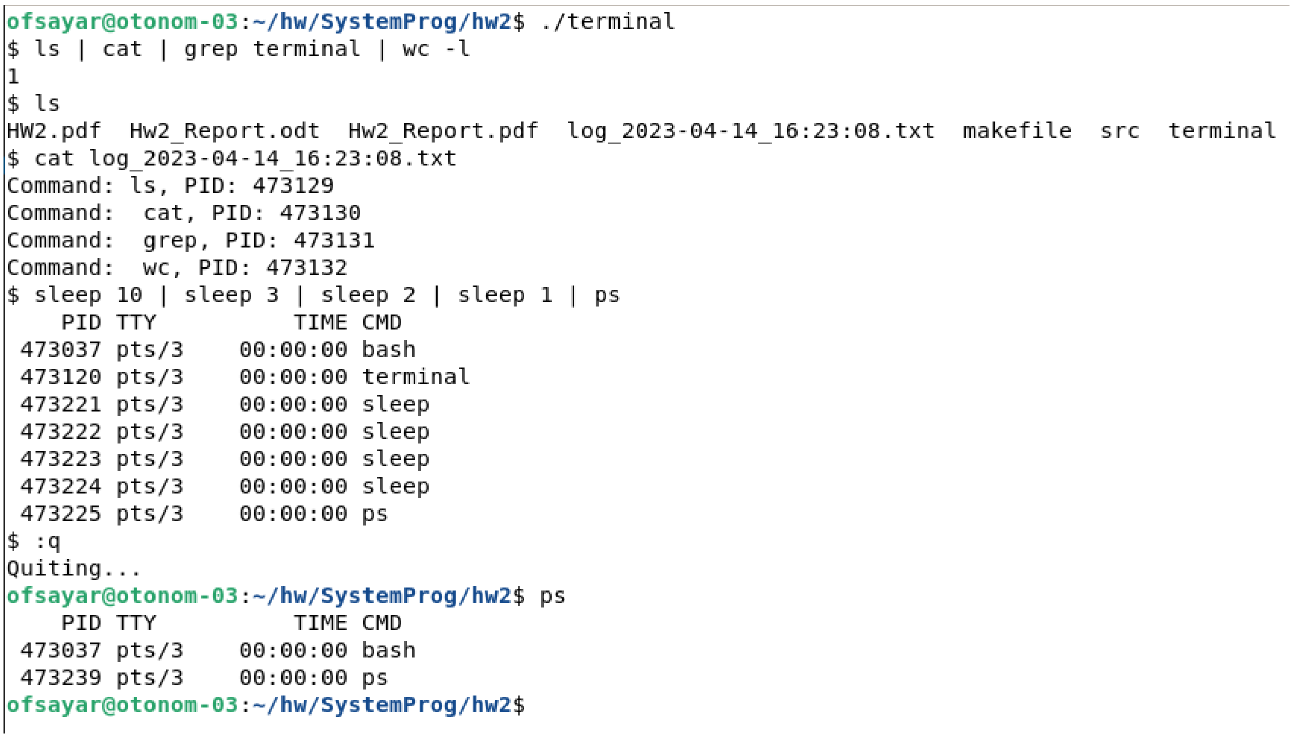
```
ofsayar@otonom-03:~/hw/SystemProg/hw2$ ./terminal
$ ls
HW2.pdf  Hw2_Report.odt  Hw2_Report.pdf  makefile  src  terminal
$ ls > a.txt
$ cat < a.txt
a.txt
HW2.pdf
Hw2_Report.odt
Hw2_Report.pdf
log_2023-04-14_16:09:54.txt
makefile
src
terminal
$ ls >
No output file specified!
$ cat <
No input file specified!
$ ps
    PID TTY          TIME CMD
 470920 pts/3    00:00:00 bash
 471807 pts/3    00:00:00 terminal
 471965 pts/3    00:00:00 ps
$
```

3) Signal handling – Zombie process check



```
ofsayar@otonom-03:~/hw/SystemProg/hw2$ make
gcc -g -Wall -pedantic-errors src/terminal_emulator.c -o terminal
ofsayar@otonom-03:~/hw/SystemProg/hw2$ ./terminal
$ ^C
Caught signal 2
$ ^Z
Caught signal 20
$ ps
    PID TTY          TIME CMD
 470920 pts/3    00:00:00 bash
 472169 pts/3    00:00:00 terminal
 472178 pts/3    00:00:00 ps
$ sleep 30 | sleep 50 | sleep 18 | sleep 42
^C
Caught signal 2, killing child processes...
$ ps
    PID TTY          TIME CMD
 470920 pts/3    00:00:00 bash
 472169 pts/3    00:00:00 terminal
 472192 pts/3    00:00:00 ps
$ sleep 100 | sleep 200 | sleep 300 | sleep 400 | kill --signal SIGQUIT 472169
Caught signal 3, quiting...
ofsayar@otonom-03:~/hw/SystemProg/hw2$ ps
    PID TTY          TIME CMD
 470920 pts/3    00:00:00 bash
 472228 pts/3    00:00:00 ps
ofsayar@otonom-03:~/hw/SystemProg/hw2$
```

4) Multiple child – log – quit



```
ofsayar@otonom-03:~/hw/SystemProg/hw2$ ./terminal
$ ls | cat | grep terminal | wc -l
1
$ ls
HW2.pdf  Hw2_Report.odt  Hw2_Report.pdf  log_2023-04-14_16:23:08.txt  makefile  src  terminal
$ cat log_2023-04-14_16:23:08.txt
Command: ls, PID: 473129
Command:  cat, PID: 473130
Command:  grep, PID: 473131
Command:  wc, PID: 473132
$ sleep 10 | sleep 3 | sleep 2 | sleep 1 | ps
    PID TTY          TIME CMD
 473037 pts/3    00:00:00 bash
 473120 pts/3    00:00:00 terminal
 473221 pts/3    00:00:00 sleep
 473222 pts/3    00:00:00 sleep
 473223 pts/3    00:00:00 sleep
 473224 pts/3    00:00:00 sleep
 473225 pts/3    00:00:00 ps
$ :q
Quiting...
ofsayar@otonom-03:~/hw/SystemProg/hw2$ ps
    PID TTY          TIME CMD
 473037 pts/3    00:00:00 bash
 473239 pts/3    00:00:00 ps
ofsayar@otonom-03:~/hw/SystemProg/hw2$
```