Gebze Technical University

Department Of Computer Engineering

CSE 344 Spring 2023

System Programming

Homework #02

Due Date: 14.04.2023

Abdurrahman Bulut
1901042258

Introduction

In this assignment, we are expected to develop a terminal emulator that can handle up to 20 shell commands in a single line without using the "system()" function from the standard C library. Instead, it is allowed to use the "fork()", "execl()", "wait()" and "exit()" functions. There are some features to be implemented:

- Each shell command should be executed via a newly created child process, meaning that multiple commands will result in multiple child processes.
- Proper handling of pipes ("|") and redirections ("<", ">") by redirecting the appropriate file descriptors.
- Usage information should be printed if the program is not called properly.
- Error messages and signals that occur during execution should be printed, and the program should return to the prompt to receive new commands.
- Aside from a SIGKILL (which also should be handled properly) the program must wait for “:q” to finalize its execution.
- Upon completion, all pids of child processes with their corresponding commands should be logged in a separate file. Each execution should create a new log file with a name corresponding to the current timestamp.
- Test the program with multiple shell commands in /bin/sh.
- It is asked to use truly functions of fork(), execl(), wait(), and exit()
- It is asked to write a makefile with “make clean”.
- It is asked to clean up after the child processes and handles with leaving zombies.
- It is asked to be careful of memory leaks.

Some Informations that given by assistant teacher Efkan Duraklı:

- Pipes ("|") are used to send the output of one command as input to another command. For example, you can use the pipe to send the output of the "ls" command to the "grep" command like below:

  - ls | grep myfile

  - This will list all files in the current directory and then filter out the output to show only files that contain the string "myfile".

- Redirections are used to change the source or destination of a command's input or output. The input redirection("<") is used to redirect input of command from a file, like below:

    - sort < myfile.txt

    - This will sort the contents of the "myfile.txt" file and output the results to the console.

- The output redirection (">") is used to redirect the output of the command to a file, like below:

    - cat file1.txt > file2.txt

    - This will write the content of "file1.txt" to "file2.txt"

- The terminal emulator doesn't need to support command separators such as ";", "&&", and "||". The requirement that "at most 20 shell commands [can be] in one line" simply means that the program should be capable of handling up to 20 shell commands using pipes or redirections. As a result, at most 20 child processes can be created.

- The example "cat > file1 | grep xxx" consists of two commands in total.

- It should handle signals such as SIGINT, and SIGTERM. The terminal emulator program can be terminated by SIGKILL signal or typing ":q". Therefore, It is needed to handle SIGKILL in addition to other signals.

- The terminal emulator program will consist of a parent process and child processes for running commands. If the program receives a signal, such as SIGINT or SIGTERM, it should terminate all child processes and display information about the signal on the screen. Afterward, the program should return to the prompt to receive new commands. However, if the program receives the SIGKILL signal, the parent process (terminal emulator program) should also be terminated.

How did I solve it?

I created a loop in the main method of my code. In there my program reads input from user until press ":q". This command triggers a kill operation of child processes. It will call "kill_child_processes" method and it prints which processes are killed in the terminal. I will explain all of the functions below. In this loop, I will print the logs of children to a ".log" file. The name of the file that was created is defined as a time variable. It parses the input that the user gives with the parse_user_input method. It assigns the command to the 2D string variable. commands variable is a triple pointer with a 2D string. I mean it is "char ***commands". Parent process saves child process info to the cmdInfo array and it increments the child process number variable After parsing the input to commands pointer, It will call run_commands function.. In this func, It forks a child process for each command. Then I set up the input and output redirections. I am using the dup method for this purpose. And finally It executes with execl function. Let's explain the code,

1. The libraries. I added those libraries to use some properties. For example, to use the "SIGINT" keyword, I needed to use signal.h library.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <time.h>
```

2. It is said that the program will be tested with a maximum of 20 shell commands. And I created 1 constant for it and 1 constant for keeping the size of user input. Each command will be a child process in this project. Then, I created a structure to store the command and its process ID. I created an array for command objects. And also a variable number of child processes is created to keep tracking the child process counts. I implemented 7 methods for this homework. All of them are implemented at the bottom of the main function except the count_args method. This method simply returns the number of arguments. I used it where I use execl function. Firstly I was using execvp but I changed my mind and I forced to use execl instead.

```c
#define MAX_COMMANDS 20
#define BUFFER_SIZE 1024

typedef struct
{
    char *cmd;
    pid_t pid;
} CmdInfo;

CmdInfo cmd_info[MAX_COMMANDS];
int num_child_processes = 0;

int count_args(char **args) {
    int count = 0;
    while (args[count] != NULL) {
        count++;
    }
    return count;
}

void kill_child_processes(int);
void signal_handler(int, siginfo_t *, void *);
void setup_signal_handlers();
char ***parse_user_input(char *input);
char **parse_cmd_args(char *command);
void run_commands(char ***, FILE *);
```

3. This is my main function. Here, I set up the signal handler and environment to use. It takes input from the user using the fgets() method and executes this input by sending to related functions. If the user enters ":q", the program will break out. run _command() executes the inputs. After breaking out, it will kill all child processes and exit the program. If the user enters an interrupt button, then it exit from the terminal but the program will be working indeed.

```c
int main()
{

    setup_signal_handlers();
    char input[BUFFER_SIZE];
    char ***commands;
    time_t t;

    while (1)
    {
        printf("terminal> ");
        fgets(input, BUFFER_SIZE, stdin);
        input[strcspn(input, "\n")] = 0;

        if (strcmp(input, ":q") == 0)
        {
            kill_child_processes(SIGTERM);
            return 0;
        }

        t = time(NULL);
        char log_filename[256];
        strftime(log_filename, sizeof(log_filename), "%Y%m%d%H%M%S.log", localtime(&t));
        FILE *log_file = fopen(log_filename, "a");

        commands = parse_user_input(input);

        run_commands(commands, log_file);
        fclose(log_file);

        for (int i = 0; i < num_child_processes; ++i)
        {
            free(commands[i]);
        }
        free(commands);
    }

    return 0;
}
```

4. This kill_child_processes function responsible for terminating the child processes and waiting for them to exit. I used WNOHANG to wait for a child to finish its job. Actually I used to simply wait but after that I cannot use “:q” correctly. Then I searched and found this solution. It is waiting all process until they exit and it is decreasing number of children one by one. By doing this I prevent zombie processes from being cerated.

```c
void kill_child_processes(int signum)
{
    for (int i = 0; i < num_child_processes; i++)
    {
        if (cmd_info[i].pid > 0)
        {
            printf("Terminating child process with Pid : %d\n", cmd_info[i].pid);
            kill(cmd_info[i].pid, signum);
        }
    }
    int status;
    pid_t pid;
    // :q çalışsın diye.  will not block if no child processes have changed state.
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0)
    {
        for (int i = 0; i < num_child_processes; i++)
        {
            if (cmd_info[i].pid == pid)
            {
                printf("Child process with PID %d exited, status code: %d\n", pid, status);
                num_child_processes--;
                break;
            }
        }
    }
}
```

5. signal_handler handles received signals. It kills child processes when called. Setup func sets up the signal handlers. I used sigaction instead of signal() method. It gives more control over the signals.

```c
void signal_handler(int signum, siginfo_t *siginfo, void *context)
{

    if (signum == SIGINT)
    {
        printf("\nReceived signal %d, stopping child processes...\n", signum);
        kill_child_processes(signum);
        return;
    }

    printf("Received signal %d, stopping child processes and exiting...\n", signum);
    kill_child_processes(signum);
    exit(EXIT_FAILURE);
}

void setup_signal_handlers()
{

    struct sigaction act;
    memset(&act, 0, sizeof(act));
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = signal_handler;

    if (sigaction(SIGINT, &act, NULL) < 0)
    {
        perror("sigaction (SIGINT) error");
        exit(1);
    }

    if (sigaction(SIGTERM, &act, NULL) < 0)
    {
        perror("sigaction (SIGTERM) error");
        exit(1);
    }
}
```

6. Parse user input method takes user's input and parse it into separate commands. It returns 3D array of string. Malloc func reserve some memory for 20 commands. I used strtok_r instead of strtok because, with strtok I couldnt take all commands. It gives first command each time. Strtok_r is thread_safe they said. Parse_cmd_args func gets 2D string array and it splits commands to arguments.

```c
char ***parse_user_input(char *input)
{
    char ***commands = malloc(20 * sizeof(char **));
    char *command;
    int index = 0;

    char *saveptr1;

    command = strtok_r(input, "|", &saveptr1);

    while (command != NULL && index < MAX_COMMANDS)
    {
        commands[index++] = parse_cmd_args(command);
        command = strtok_r(NULL, "|", &saveptr1);
    }

    if (index <= 20)
    {
        num_child_processes = index;
    }

    return commands;
}
char **parse_cmd_args(char *command)
{

    char **args = malloc(64 * sizeof(char *));
    char *arg;
    int index = 0;

    arg = strtok(command, " ");
    while (arg != NULL)
    {
        args[index++] = arg;
        arg = strtok(NULL, " ");
    }

    args[index] = NULL;

    return args;
}
```

7. This func takes an array of commands and log file as input. Since each command is a child process, it will prints command name and pid of child process to a log file. I printed a command line to a log file. For example, ls | ps, will be written to a single log file. It initialize file descriptors for input and output and sets them to current command. It creates a pipe to connect the commands. It firks the process to create a child process. In the child process, it adjust the file descriptors for input and output usind dup2(). It handles redirections with “<”, “>”. If the process it parent, then it will waits for child process is finished using waitpid(). And it clean process id’s and file descriptors.

```c
void run_commands(char ***commands, FILE *log_file)
{
    int i;
    int in_fd = 0;
    int out_fd = 1;
    int pipe_fd[2];

    for (i = 0; i < num_child_processes; ++i)
    {
        char **args = commands[i];

        if (i != 0)
        {
            in_fd = pipe_fd[0];
        }

        if (i != num_child_processes - 1)
        {
            pipe(pipe_fd);
            out_fd = pipe_fd[1];
        }
        else
        {
            out_fd = 1;
        }

        pid_t pid = fork();
        if (pid == 0)
        {
            if (in_fd != 0)
            {
                dup2(in_fd, 0);
                close(in_fd);
            }

            if (out_fd != 1)
            {
                dup2(out_fd, 1);
                close(out_fd);
            }

            for (int j = 0; args[j]; ++j)
            {
                if (strcmp(args[j], "<") == 0)
                {
                    int file_fd = open(args[j + 1], O_RDONLY);
```

```
                if (file_fd == -1)
                {
                    perror("Error opening input file");
                    exit(EXIT_FAILURE);
                }

                dup2(file_fd, 0);
                args[j] = NULL;
                close(file_fd);
            }
            else if (strcmp(args[j], ">") == 0)
            {
                int file_fd = open(args[j + 1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
                if (file_fd == -1)
                {
                    perror("Error opening output file");
                    exit(EXIT_FAILURE);
                }
                dup2(file_fd, 1);
                args[j] = NULL;
                close(file_fd);
            }
        }

        // execvp kullanmak yerine
        int arg_count = count_args(args);
        char *cmd_str = NULL;
        int cmd_str_len = 0;
        for (int j = 0; j < arg_count; ++j) {
            cmd_str_len += strlen(args[j]) + 1;
            cmd_str = realloc(cmd_str, cmd_str_len);
            if (j == 0) {
                strcpy(cmd_str, args[j]);
            } else {
                strcat(cmd_str, " ");
                strcat(cmd_str, args[j]);
            }
        }

        if (execl("/bin/sh", "sh", "-c", cmd_str, (char *)NULL) == -1) {
            perror("Error executing command");
            exit(EXIT_FAILURE);
        }
        free(cmd_str);
```

```
        }
        free(cmd_str);
    }
    else if (pid < 0)
    {
        perror("Error forking");
    }
    else
    {
        cmd_info[i].cmd = args[0];
        cmd_info[i].pid = pid;

        int status;
        waitpid(pid, &status, 0);

        fprintf(log_file, "command: %s , pid: %d\n", args[0], pid);

        if (in_fd != 0)
        {
            close(in_fd);
        }

        if (out_fd != 1)
        {
            close(out_fd);
        }
    }
}
}
```

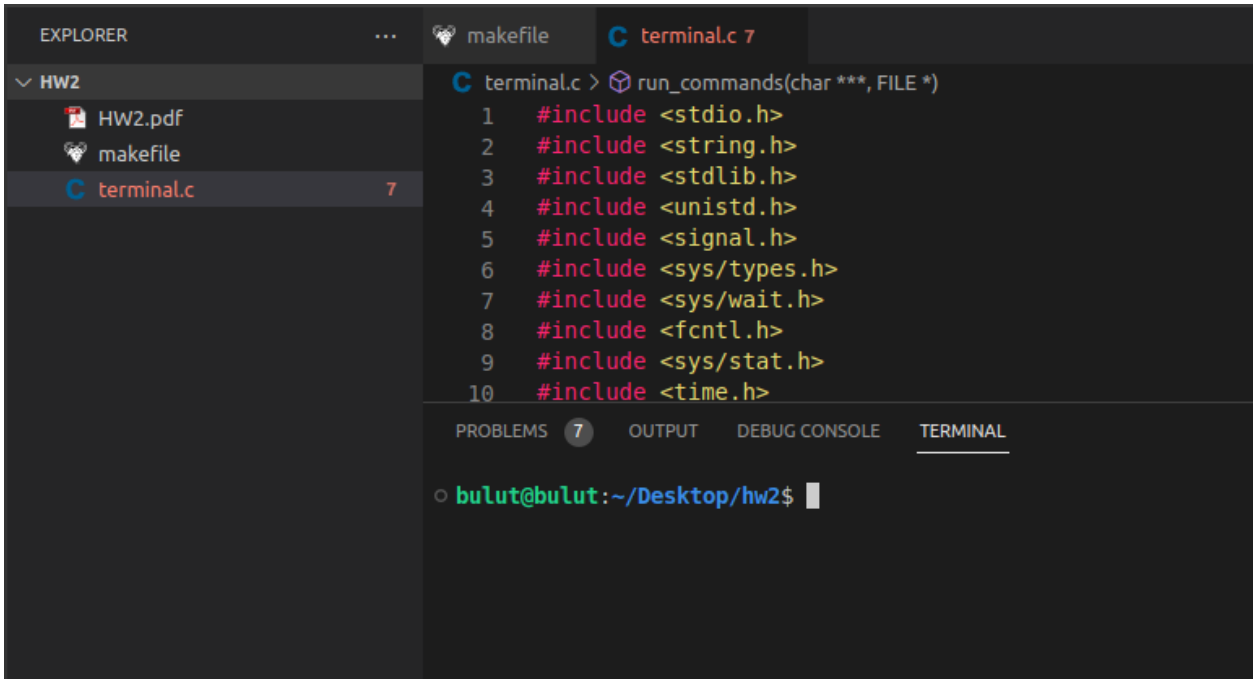
Which requirements I met?

- I used fork(), execl(), wait() and exit() correctly.
    - I am using fork to create a child for each command.
    - I am using execl in the child process.It replace the child process with new process image.
    - I am using wait. With this the parent waits for child process complete.
    - I am using exit to terminate.

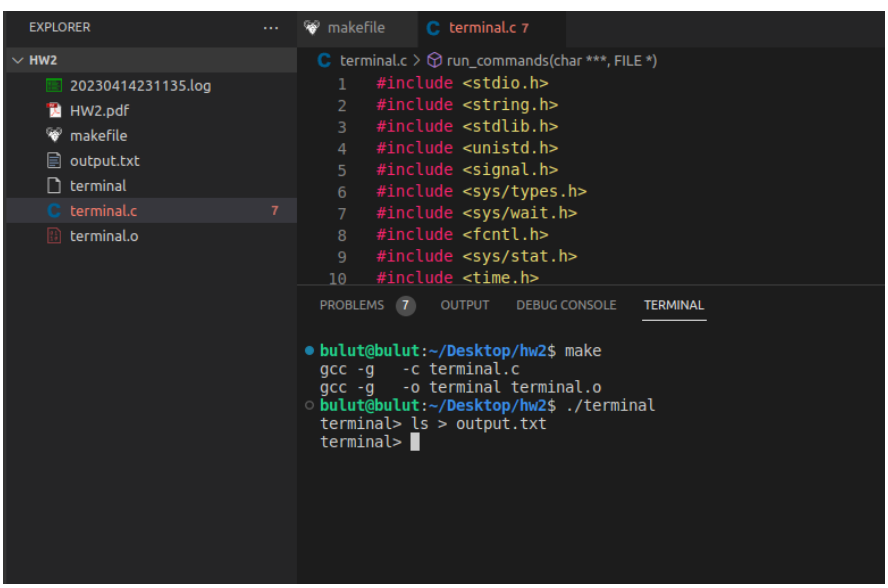
- I am correctly clean up after child processes with kill_child_process method.
- I also prevent zombie processes from being created with this. I will show some tests for it.
- I checked the memory leaks with valgrid and it seems there is no any problem.
- I am using pipe and redirections. I will show some tests.
- I created a makefile with “make clean”
- I suppose I completed all requirements.
- It prints child processes info to a log file for each command line. It uses a log file for one time commands. For example, ls | ps , two of them will be written to a log file. Another command group will be written to another one log file.
- Terminated child info is printed to console.

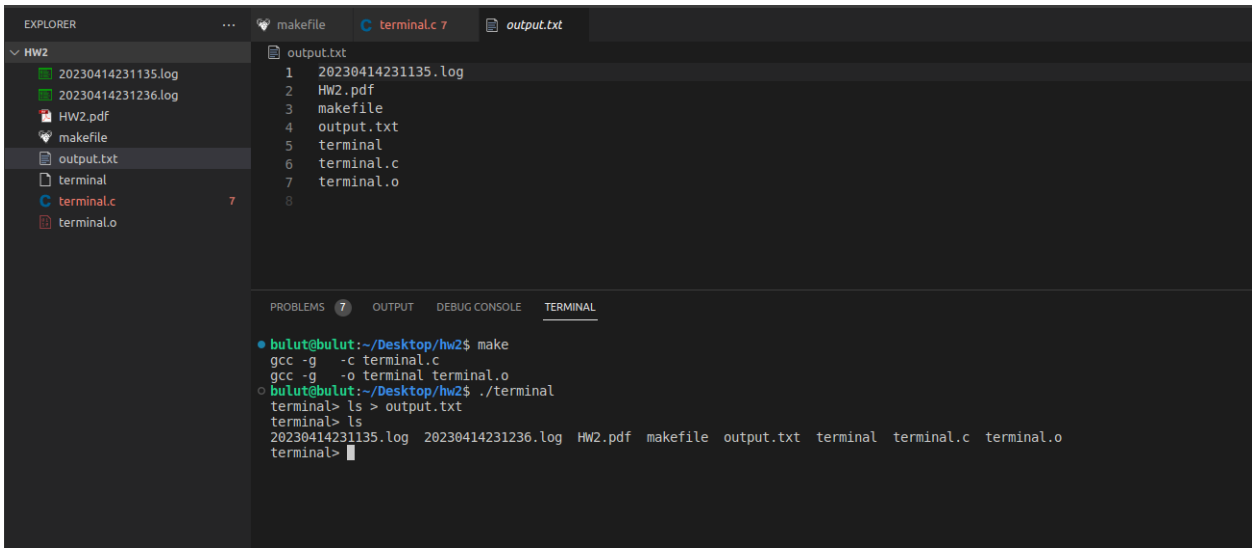
Tests

File structure at the beginning.

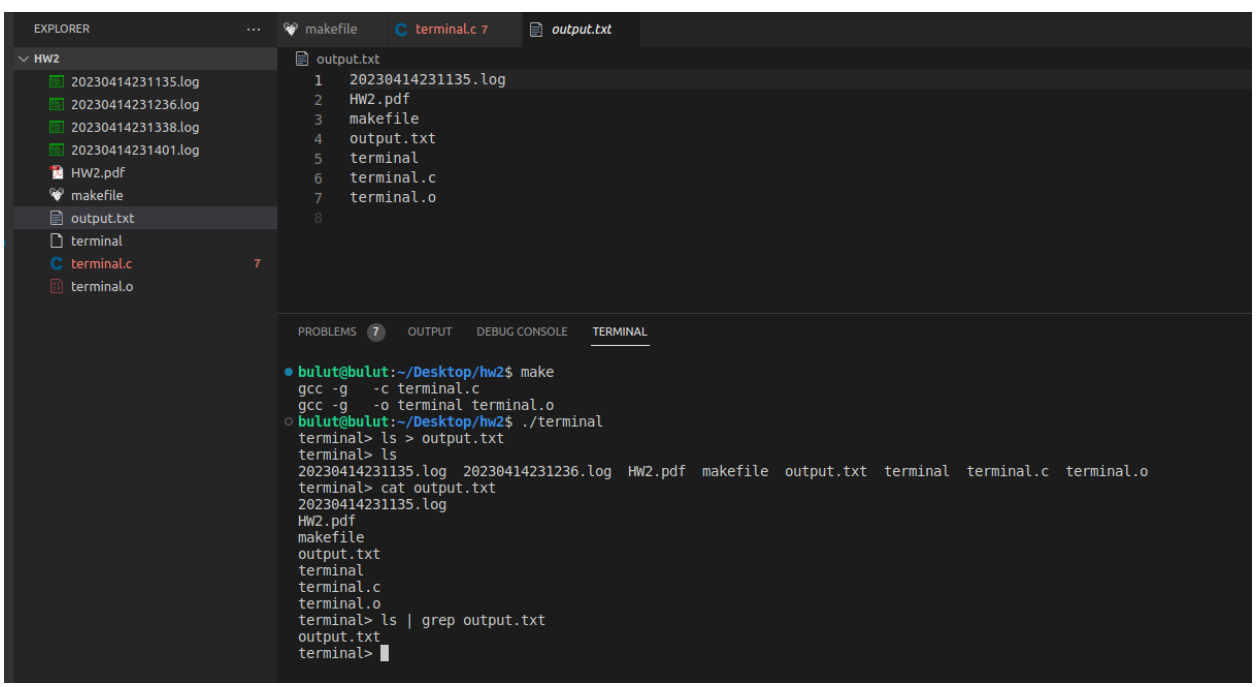


- **Test 1:Redirections and Pipes**
- It creates output file and wrote the ls result to it.

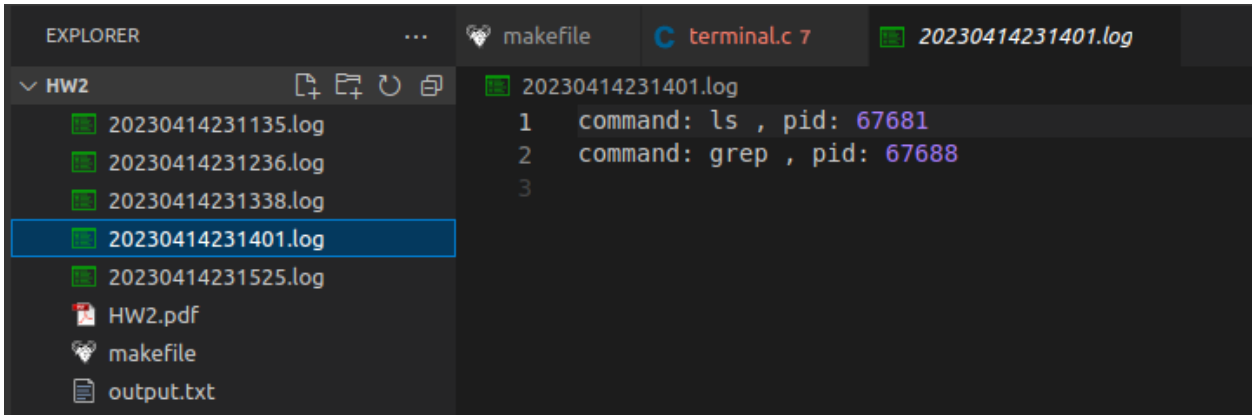




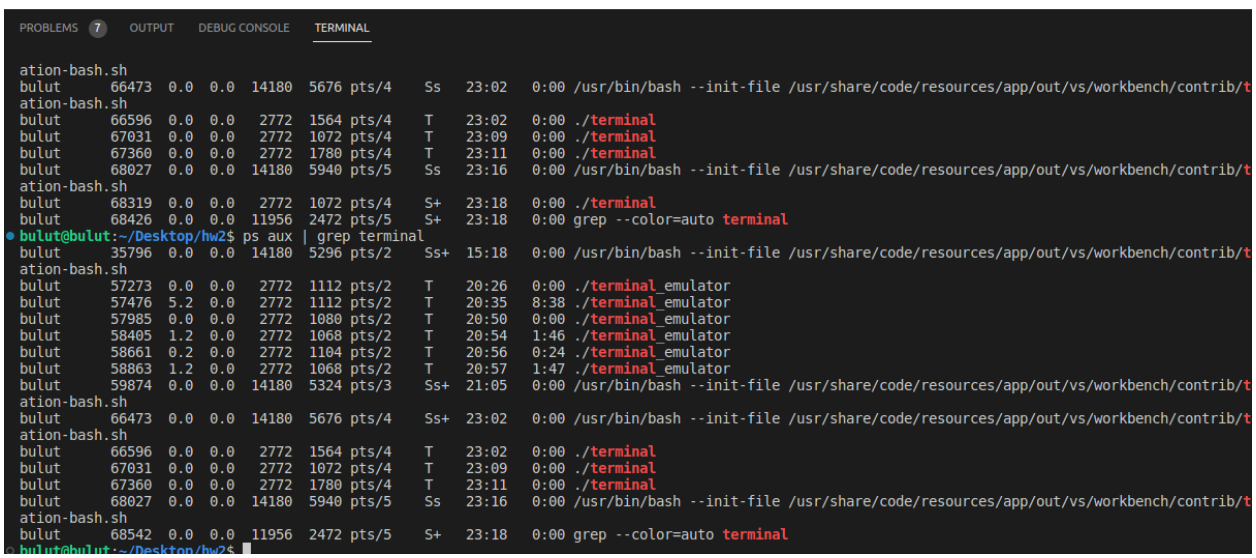
- **Test 2: test the pipe functionality. ls | grep output.txt**



And log files:



- Test cleaning up child processes
- ps aux | grep terminal
- After doing ":q" operation and exit from the program, They will be gone.



- **Test 3: memory leak (valgrind)**
- I used this command “valgrind --leak-check=full ./terminal”

```
bulut@bulut:~/Desktop/hw2$ valgrind --leak-check=full ./terminal
==68717== Memcheck, a memory error detector
==68717== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==68717== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==68717== Command: ./terminal
==68717==
terminal> ps
    PID TTY          TIME CMD
  68027 pts/5    00:00:00 bash
  68717 pts/5    00:00:00 memcheck-amd64-
  68738 pts/5    00:00:00 sh
  68739 pts/5    00:00:00 ps
terminal> ls
20230414231135.log  20230414231338.log  20230414231525.log  20230414231609.log  20230414231806.log  20230414232059.log  makefile    terminal    terminal.o
20230414231236.log  20230414231401.log  20230414231608.log  20230414231805.log  20230414232058.log  HW2.pdf             output.txt  terminal.c
terminal> mkdir new
terminal> touch new.txt
terminal> :q
Terminating child process with Pid : 68786
==68717==
==68717== HEAP SUMMARY:
==68717==     in use at exit: 0 bytes in 0 blocks
==68717==   total heap usage: 31 allocs, 31 frees, 28,922 bytes allocated
==68717==
==68717== All heap blocks were freed -- no leaks are possible
==68717==
==68717== For lists of detected and suppressed errors, rerun with: -s
==68717== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
bulut@bulut:~/Desktop/hw2$
```

It says All heap blocks were freed – **no leaks are possible.**