Ana Clara Mueller Miranda - RA 148806
Bruno Pires Moreira Silva - RA 139892
Igor Ribeiro Ferreira de Matos - RA 140492
Rafaela Cristine dos Santos Uchôas - RA 140351

Relatório Projeto 1 - Shell

1. Sobre o projeto

O trabalho desenvolvido em questão é basicamente o protótipo de um interpretador de comandos Linux, também conhecido como Shell, na linguagem C, capaz de executar comandos conforme os quatro requisitos descritos na definição do exercício (executar comandos unitários, com múltiplos parâmetros, executar comandos encadeados com o operador pipe, executar comandos condicionados e executar comandos em background).

Especificando mais sobre o código desenvolvido, 6 funções foram feitas, além da main que as utiliza dessas funções, de modo que, 3 funções são para o tratamento de tokens (com tokens, queremos dizer os operadores responsáveis por, de alguma forma, concatenar comandos. Mais especificamente, no nosso caso, token seria o operador pipe “|”, OR “||” , AND “&&” e o operador para executar comentos em background, “&”). Essas três funções são:

- `conta_tk` - Responsável pela contagem de tokens, dada a matriz de comando argv. Note que ela é utilizada apenas para identificar a quantidade de comandos encadeados, reconhecendo o operador pipe.

```c
int conta_tk(int argc, char **argv, char *tk)
{
    int i = 0, qtde = 0;
    char *token = NULL;
    for (i = 0; i < argc; i++) // conta a quantidade de pipe (|)
    {
        token = strtok(argv[i], tk);
        if (token == NULL)
        {
            qtde++;
        }
    }
    return qtde;
}
```

- `encontra_tks` - Responsável por retornar o índice de tokens que condicionam comandos, no caso do OR ou AND, dada a matriz de argumentos.
- `encontra_tk` - Responsável por retornar uma flag se existe ou não um operador com caracter único, que é útil para o reconhecimento de comandos de background, uma

vez que o reconhecimento de todos os outros tipos de comandos já foram tratados.

```c
int encontra_tks(int argc, char **argv, char *tk1, char *tk2)
{
    int token_or = -1, token_and = -1, i = 0;
    for (i = 0; i < argc; i++) // procura operadores condicionais "e" (&&) ou "ou" (||)
    {
        token_or = strcmp(argv[i], tk1);
        token_and = strcmp(argv[i], tk2);
        if (token_or == 0 || token_and == 0)
        {
            return i;
        }
    }
    return -1;
}
```

- `comando_simples` - Responsável por executar comandos simples, únicos, sem condicionais ou encadeamentos. A árvore de processos é diretamente instanciada, com os processos filhos criados se necessários ou com a execução direta do comando, de acordo com o retorno da função fork()

```c
int comando_simples(char **cmd, int bg)
{
    pid_t pid; // instancia o indentificador do processo

    pid = fork();
    if (pid == 0) // processo filho
    {
        execvp(cmd[0], cmd);
        return 0;
    }
    else if (pid > 0) // processo pai
    {
        int status;
        if (!bg) // se FALSE, comando não tem bg, então pai aguarda filho
        {
            waitpid(pid, &status, 0);
        } // senão, pai há bg, então pai não espera filho
        return WEXITSTATUS(status);
    }
    else
    {
        return -1;
    }
}
```

- `comando_pipes` - Responsável por tratar de comandos encadeados, o qual a variável `int fd` representa o buffer, ou seja, o pipe em si. A árvore de processos é criada, e os processos são executados de acordo com a quantidade de pipes existentes, como estabelece o laço externo, que itera reconhecendo os comandos através do operador pipe, e o limite desse laço é exatamente o número de pipes reconhecidos, que é um dos argumentos dessa própria função. Os processos filhos são executados enquanto os processos pai estão em sleep, sobrescritos em memória, sendo que esse comportamento é definido pelas funções `execvp()` e `waitpid()`.

```c
int comando_pipes(int argc, char **argv, int qtde_pipes)
{
    int fd[2], i = 0, pos_pipe; // fd[2] representa a criação de dois canais de comunicação, 1 par
    int aux = STDIN_FILENO;

    for (int j = 0; j <= qtde_pipes; j++)
    {
        // formata o comando atual de acordo com a posicao do pipe retornada
        char **cmd = &argv[i]; // copia o comando referente a posição na matriz de argumento
        pos_pipe = encontra_pipe(argc, cmd, "|");

        // argv[pos_pipe]=NULL;
        // printf("i: %d, pos pipe: %d\n", i, pos_pipe);
        // printf("cmd: %s %s\n", cmd[0], cmd[1]);
        cmd[pos_pipe] = NULL; //adiciona NULL onde teria o pipe, igual é feito na função encontrat
        if (qtde_pipes != -1)
            cmd[pos_pipe - i] = NULL;


        if (pipe(fd) < 0)
        {
            return -1;
        }

        pid_t pid = fork();

        if (pid == 0)   //processo filho
        {
            close(fd[0]);
            dup2(aux, STDIN_FILENO);
            if (j < qtde_pipes)
                dup2(fd[1], STDOUT_FILENO); // duplica saida padrao do filho pa

            execvp(cmd[0], cmd); // fornece um vetor de ponteiros representando
            return 0;
        }
        else if (pid > 0)
        { // processo pai
            aux = fd[0];
            close(fd[1]); // pai nao vai escrever
            waitpid(pid, NULL, 0);
        }
        else
        {
            perror("fork");
            return -1;
        }

        i = i + pos_pipe +1;
    }

    return 1;
}
```

- `comando_cond` - Responsável por executar comandos condicionais, além de checar essas próproas condições primeiramente. Um detalhe importante é que essa função é chamada recursivamente, uma vez que os comandos são checados dois a dois, de acordo com o condicional (OR ou AND). Como ela as comandos em si são não

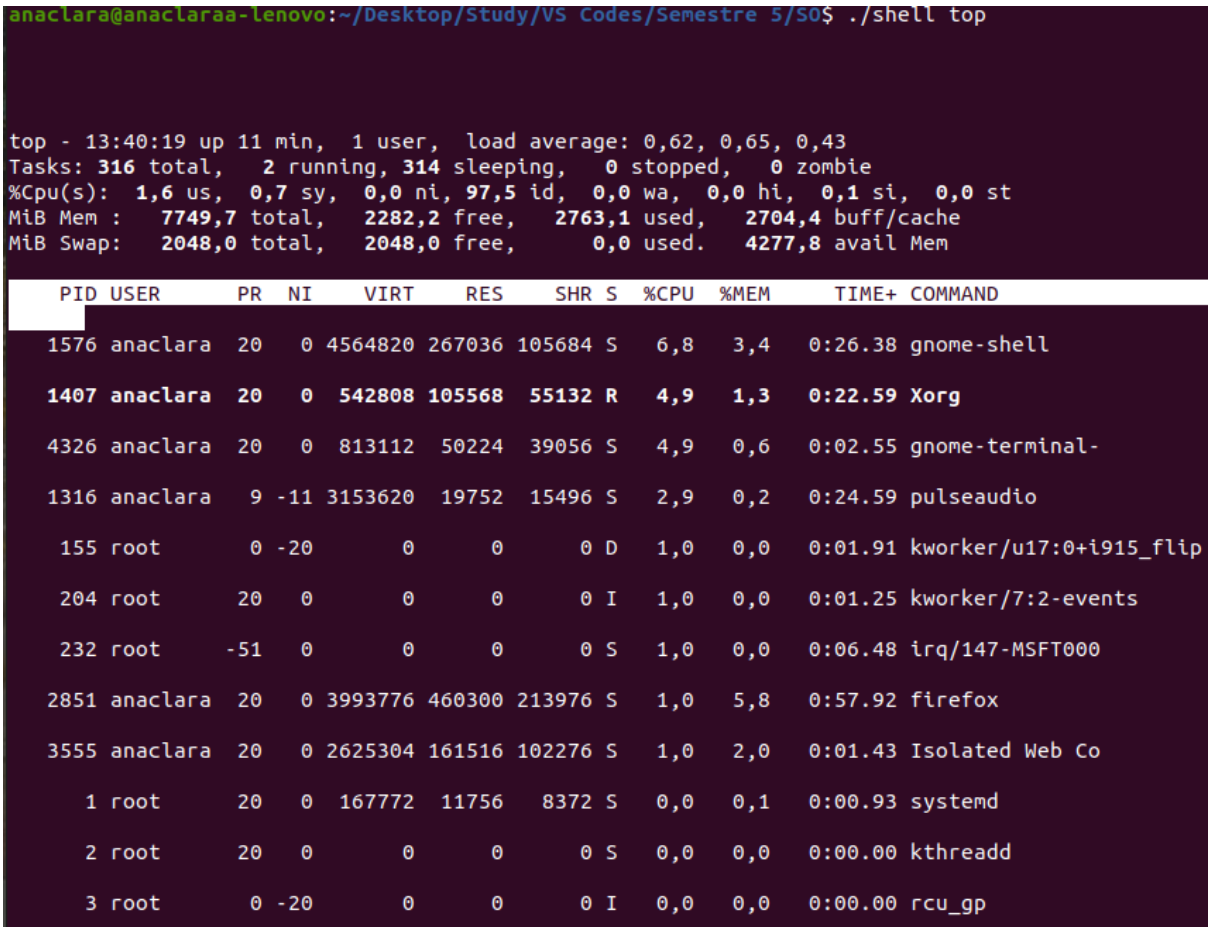
deixam de ser comandos simples, unitários, a função `comando_simples` é chamada para a execução deles. A chamada recursiva se dá com base na condição, uma vez que nem sempre ela precisará ser executada

```c
int comando_cond(int argc, char **cmd, int pos){
    char **cond_cmd;
    int current_condition=0;

    if (pos != -1)
    {
        // formata o comando atual e o prox comando corretamente
        cond_cmd = &cmd[pos];
        // printf("pos: %d - %s, %s\n", pos, cmd[pos], cond_cmd[0]);
        if(!strcmp(cmd[pos-1],"&&")){
            // printf("\ne\n");
            current_condition=1; //e
        }else{
            // printf("\nou\n");
            current_condition=2; //ou
        }
        cmd[pos-1] = NULL;
    }
    else
    {
        current_condition = 0;
    }
```

- Função main - Como de costume, recebe a o número de argumentos e a matriz de comandos. Basicamente, sua função é reconhecer o tipo de comando que está sendo tratado e chama a função correspondente.
- Exemplo de entrada e saída:

```
anaclara@anaclaraa-lenovo:~/Desktop/Study/VS Codes/Semestre 5/SO$ ./shell3 ls -la
total 92
drwxrwxr-x 2 anaclara anaclara  4096 mai 16 13:37  .
drwxrwxr-x 5 anaclara anaclara  4096 mai 12 13:28  ..
-rw-rw-r-- 1 anaclara anaclara   628 mai  3 14:32 'Link dos arquivos - Shell.txt'
-rwxrwxr-x 1 anaclara anaclara 17464 mai  3 16:31  shell
-rw-rw-r-- 1 anaclara anaclara  3673 abr 28 16:26  shell1.c
-rwxrwxr-x 1 anaclara anaclara 17440 mai 16 13:37  shell3
-rw-rw-r-- 1 anaclara anaclara  6955 mai 16 10:15  shell3.c
-rw-rw-r-- 1 anaclara anaclara  7435 mai  3 16:27  shell.c
```



2. Organização do grupo

Após discussões para alinhar todo o grupo da tarefa, tentando equilibrar o conhecimento de todos acerca do que era necessário, foi definido o seguinte, sendo que todos os tópicos foram discutidos em grupo antes de serem divididos:

- Função para comandos com pipe e background: Rafaela
- Funções de comandos simples e condicionais: Igor
- Funções de tratamento da string de comando e função main: Bruno Pires e Ana Clara

3. Auto avaliação do grupo

Embora talvez a divisão de tarefas e os testes do código não foram feitas da melhor forma possível, pois houve a impressão de que alguns membros ficariam sobrecarregados com algumas das funções feitas e foi necessário a ajuda do resto do grupo, assim como nem todos os casos foram forçados a fim de estressar ao máximo o código e garantir que não há casos de erros, e considerando também as dificuldades pessoais de cada um dos integrantes que refletem na totalidade do grupo, podemos dizer que para um primeiro

trabalho, a auto avaliação pode ser dita como satisfatória, com pontos a melhorar no próximo projeto