```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <string.h>
#include <dirent.h>
#include <pwd.h>

void printTopInfo() {
    //Abre o ficheiro no /proc/loadavg para obter as informacoes acerca da carga
media do CPU
    FILE *loadavg_file = fopen("/proc/loadavg", "r");

    //Verifica se o ficheiro /proc/loadavg existe e o seu processo foi bem-
sucedido
    if (loadavg_file == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    //Variaveis relativas a guardar as informacoes acerca a carga media do CPU
    double loadavg_1min, loadavg_5min, loadavg_15min;

    //Associa cada valor respetivamente a informacao que transmite
    fscanf(loadavg_file, "%lf %lf %lf", &loadavg_1min, &loadavg_5min,
&loadavg_15min);

    //Fecha o ficheiro no /proc/loadavg
    fclose(loadavg_file);

    //Mostra ao utilizador as informacoes acerca da carga media do CPU para os
minutos 1, 5 e 15 minutos
    printf("Carga media do CPU: %.2f %.2f %.2f\n", loadavg_1min, loadavg_5min,
loadavg_15min);

    //Variaveis que armazenam o numero total de processos e os em execucao.
    int total_processes = 0, running_processes = 0;

    //Abre o diretorio /proc para obter informacoes sobre os processos
    DIR *proc_dir = opendir("/proc");

    //Verifica se o diretorio /proc existe e o seu processo foi bem-sucedido
    if (proc_dir == NULL) {
        perror("opendir");
        exit(EXIT_FAILURE);
    }

    //Estrutura que guarda informacoes relativas ao diretorio, bem como o nome
de diretorios ou ficheiros presentes nesta (d_name)
    //ino_t d_ino; - Número do inode (Armazena informacoes como: proprietário,
permissoes (r-w-e), entre outros..)
    //off_t d_off; - Deslocamento (Armazena o deslocamento para a proxima
entrada no diretorio)
    //unsigned short d_reclen; - Tamanho desta estrutura (Armazena o tamanho da
estrutura de dados da entrada do diretorio)
    //unsigned char d_type; - Tipo (Armazena o tipo da entrada do diretorio
como: DT_REG (arquivo) e DT_DIR (diretorio))
    //char d_name[]; - Nome (Array de caracteres que armazena o nome)
    struct dirent *entry;

    //Loop para percorrer todos os diretorios de processos
    while ((entry = readdir(proc_dir)) != NULL) {
```

```
        //Verifica se o nome do diretorio é um número (PID)
        if (atoi(entry->d_name) > 0) {
            total_processes++;

            //Criacao do caminho para o ficheiro "status" de cada processo
            char proc_path[512];
            if (snprintf(proc_path, sizeof(proc_path), "/proc/%s/status", entry-
>d_name) >= sizeof(proc_path)) {
                fprintf(stderr, "Erro: Caminho do processo incorreto!\n");
                exit(EXIT_FAILURE);
            }

            //Abre o ficheiro /proc/[PID]/status para obter o estado do processo
            FILE *status_file = fopen(proc_path, "r");

            //Verifica se o ficheiro /proc/[PID]/status existe e o seu processo
foi bem-sucedido
            if (status_file == NULL) {
                perror("fopen");
                exit(EXIT_FAILURE);
            }

            //Obtem o estado de cada processo e consoante este incrementa a
variavel "running_processes"
            char status[256];
            while (fscanf(status_file, "%*s %s", status) == 1) {
                if (strcmp(status, "R") == 0) {
                    running_processes++;
                    break;
                }
            }

            //Fecha o ficheiro no /proc/[PID]/status
            fclose(status_file);
        }
    }

    //Fecha o diretorio /proc onde se obtem as informacoes sobre os processos
    closedir(proc_dir);

    //Mostra as informacoes obtidas sobre o total de processos e em execucao
    printf("Total de processos: %d\nProcessos em execução: %d\n",
total_processes, running_processes);

    //Abre novamente o diretorio /proc para obter informacoes detalhadas sobre
os processos
    DIR *proc_dir_running = opendir("/proc");

    //Verifica se o diretorio /proc existe e o seu processo foi bem-sucedido
    if (proc_dir_running == NULL) {
        perror("opendir");
        exit(EXIT_FAILURE);
    }

    //Exibe o titulo do comando "top", bem como o nome de cada coluna consoante
a informacao desta
    printf("\nProcessos\n");
    printf("%-8s %-8s %-20s %s\n", "PID", "STATE", "USERNAME", "COMMAND");

    //Variaveis relativas a quantidade de processos em execucao e nao
    int count_running_processes = 0, count_non_running_processes = 0;

    //Volta ao inicio do diretorio de processos para percorre-lo novamente
    rewinddir(proc_dir_running);
```

```
    //Loop para percorrer todos os diretorios de processos
    while ((entry = readdir(proc_dir_running)) != NULL &&
(count_running_processes + count_non_running_processes) < 20) {
        //Verifica se o nome do diretorio é um número (PID)
        if (atoi(entry->d_name) > 0) {
            //Criacao do caminho para o ficheiro "status" de cada processo
            char proc_path[512];
            if (snprintf(proc_path, sizeof(proc_path), "/proc/%s/status", entry-
>d_name) >= sizeof(proc_path)) {
                fprintf(stderr, "Erro: Caminho do processo incompleto!\n");
                exit(EXIT_FAILURE);
            }

            //Abre o ficheiro /proc/[PID]/status para obter o estado do processo
            FILE *status_file = fopen(proc_path, "r");

            //Verifica se o ficheiro /proc/[PID]/status existe e o seu processo
foi bem-sucedido
            if (status_file == NULL) {
                perror("fopen");
                exit(EXIT_FAILURE);
            }

            //Variaveis para armazenar o estado, o username e o comando do
processo
            char status[256], uid[256], username[256], cmdline[512];
            //Obtem o estado do processo
            while (fscanf(status_file, "%*s %s", status) == 1) {
                if (strlen(status) == 1 && strcmp(status, "R") == 0) {
                    // Obtem o UID (User ID) de cada processo
                    fscanf(status_file, "%*s %s", uid);

                    //Converte o UID para obter o nome do usuario de cada
processo
                    struct passwd *pwd = getpwuid(atoi(uid));

                    //Copia o nome do usuario para a variável username
                    if (pwd != NULL) {
                        strncpy(username, pwd->pw_name, sizeof(username) - 1);
                        username[sizeof(username) - 1] = '\0';  //Indicar onde
terminar a string (nome de usuario)
                    } else {
                        // Se pwd for NULL, definimos como "Unknown"
                        strncpy(username, "Unknown", sizeof(username) - 1);
                        username[sizeof(username) - 1] = '\0';  //Indicar onde
terminar a string (nome de usuario)
                    }

                    //Criacao do caminho para o ficheiro "cmdline" de cada
processo
                    if (snprintf(proc_path, sizeof(proc_path),
"/proc/%s/cmdline", entry->d_name) >= sizeof(proc_path)) {
                        fprintf(stderr, "Erro: Caminho do processo incompleto!\
n");
                        exit(EXIT_FAILURE);
                    }

                    //Abre o ficheiro /proc/[PID]/cmdline para obter o estado do
processo
                    FILE *cmdline_file = fopen(proc_path, "r");

                    //Verifica se o ficheiro /proc/[PID]/cmdline existe e o seu
processo foi bem-sucedido
                    if (cmdline_file == NULL) {
```

```
                        perror("fopen");
                        exit(EXIT_FAILURE);
                    }

                    //Obtem a linha de comando de cada processo
                    fgets(cmdline, sizeof(cmdline), cmdline_file);

                    //Fecha o ficheiro no /proc/[PID]/cmdline
                    fclose(cmdline_file);

                    //Imprime as informações detalhadas do processo conforme as
colunas
                    printf("%-8s %-8s %-20s %s\n", entry->d_name, status,
username, cmdline);

                    //Incrementa a quantidade de processos em execucao ja a
serem exibidos
                    count_running_processes++;
                    break;
                }
            }

            //Fecha o ficheiro no /proc/[PID]/status
            fclose(status_file);
        }
    }

    //Volta ao inicio do diretorio de processos para percorre-lo novamente
    rewinddir(proc_dir_running);
    //Loop para percorrer todos os diretorios de processos
    while ((entry = readdir(proc_dir_running)) != NULL &&
count_non_running_processes < (20 - count_running_processes)) {
        //Verifica se o nome do diretorio é um número (PID)
        if (atoi(entry->d_name) > 0) {
            //Criacao do caminho para o ficheiro "status" de cada processo
            char proc_path[512];
            if (snprintf(proc_path, sizeof(proc_path), "/proc/%s/status", entry-
>d_name) >= sizeof(proc_path)) {
                fprintf(stderr, "Erro: Caminho do processo incompleto!\n");
                exit(EXIT_FAILURE);
            }

            //Abre o ficheiro /proc/[PID]/status para obter o estado do processo
            FILE *status_file = fopen(proc_path, "r");

            //Verifica se o ficheiro /proc/[PID]/status existe e o seu processo
foi bem-sucedido
            if (status_file == NULL) {
                perror("fopen");
                exit(EXIT_FAILURE);
            }

            //Variaveis para armazenar o estado, o username e o comando do
processo
            char status[256], uid[256], username[256], cmdline[512];
            //Obtem o estado do processo
            while (fscanf(status_file, "%*s %s", status) == 1) {
                //S: Sleeping
                //D: Uninterruptible Sleep
                //Z: Zombie
                //T: Stopped
                //I: Idle
                if (strlen(status) == 1 && strchr("SDZTI", status[0]) != NULL) {
                    // Obtem o UID (User ID) de cada processo
```

```
                    fscanf(status_file, "%*s %s", uid);

                    //Converte o UID para obter o nome do usuario de cada
processo
                    struct passwd *pwd = getpwuid(atoi(uid));

                    //Copia o nome do usuario para a variável username
                    if (pwd != NULL) {
                        strncpy(username, pwd->pw_name, sizeof(username) - 1);
                        username[sizeof(username) - 1] = '\0';  //Indicar onde
terminar a string (nome de usuario)
                    } else {
                        // Se pwd for NULL, definimos como "Unknown"
                        strncpy(username, "Unknown", sizeof(username) - 1);
                        username[sizeof(username) - 1] = '\0';  //Indicar onde
terminar a string (nome de usuario)
                    }

                    //Criacao do caminho para o ficheiro "cmdline" de cada
processo
                    if (snprintf(proc_path, sizeof(proc_path),
"/proc/%s/cmdline", entry->d_name) >= sizeof(proc_path)) {
                        fprintf(stderr, "Erro: Caminho do processo incompleto!\
n");
                        exit(EXIT_FAILURE);
                    }

                    //Abre o ficheiro /proc/[PID]/cmdline para obter o estado do
processo
                    FILE *cmdline_file = fopen(proc_path, "r");

                    //Verifica se o ficheiro /proc/[PID]/cmdline existe e o seu
processo foi bem-sucedido
                    if (cmdline_file == NULL) {
                        perror("fopen");
                        exit(EXIT_FAILURE);
                    }

                    //Obtem a linha de comando de cada processo
                    fgets(cmdline, sizeof(cmdline), cmdline_file);

                    //Fecha o ficheiro no /proc/[PID]/cmdline
                    fclose(cmdline_file);

                    //Imprime as informações detalhadas do processo conforme as
colunas
                    printf("%-8s %-8s %-20s %s\n", entry->d_name, status,
username, cmdline);

                    //Incrementa a quantidade de processos em outros estados sem
ser execucao ja a serem exibidos
                    count_non_running_processes++;
                    break;
                }
            }

            //Fecha o ficheiro no /proc/[PID]/status
            fclose(status_file);
        }
    }

    //Fecha o diretorio /proc onde se obtem as informacoes sobre os processos
    closedir(proc_dir_running);
}
```

```
void executeCommandTop() {
    char input;

    //Enquanto que o input introduzido pelo utilizador for diferente de "q",
repete as instrucoes dentro do "do"
    do {
        //Limpa a consola
        system("clear");

        //Chama a funcao relativa as informacoes sobre o sistema e os processos
        printTopInfo();

        //Aguarda 10 segundos antes de atualizar as informações
        sleep(10);

        //Mostra mensagem para terminar ou continuar a apresentar essas
informacoes
        printf("Insira 'q' para terminar ou outra tecla para continuar: ");

        //Le o caracter introduzido pelo utilizador
        scanf("%c", &input);
    } while (input != 'q');
}

void executeCommand(char *cmd, char *args[]) {
    pid_t pid = fork();

    if (pid == 0) {
        //Código do processo filho

        //Executa o comando
        execvp(cmd, args);

        //Se o exec falhar, mostra uma mensagem de erro
        perror("execvp");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        //Código do processo pai
        int status;
        waitpid(pid, &status, 0);

        if (WIFEXITED(status) && WEXITSTATUS(status) != 0) {
            fprintf(stderr, "O comando '%s' falhou com o código de saída %d\n",
cmd, WEXITSTATUS(status));
            exit(EXIT_FAILURE);
        }
    } else {
        //Se o fork falhar, mostra uma mensagem de erro
        perror("fork");
        exit(EXIT_FAILURE);
    }
}

void executeCommandWithInputRedirection(char *cmd, char **args, char *inputFile)
{
    pid_t pid = fork();

    if (pid == 0) {
        //Código do processo filho

        //Abre o descritor do ficheiro no modo "read"
        int fd = open(inputFile, O_RDONLY);
        if (fd == -1) {
```

```
            perror("open");
            exit(EXIT_FAILURE);
        }

        //Redireciona a entrada do ficheiro
        if (dup2(fd, STDIN_FILENO) == -1) {
            perror("dup2");
            close(fd);
            exit(EXIT_FAILURE);
        }

        //Fecha o descritor não necessario apos a duplicacao
        close(fd);

        //Executa o comando
        execvp(cmd, args);

        //Se o exec falhar, mostra uma mensagem de erro
        perror("execvp");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        //Código do processo pai
        int status;
        waitpid(pid, &status, 0);

        if (WIFEXITED(status) && WEXITSTATUS(status) != 0) {
            fprintf(stderr, "O comando '%s' falhou com o código de saída %d\n",
cmd, WEXITSTATUS(status));
            exit(EXIT_FAILURE);
        }
    } else {
        //Se o fork falhar, mostra uma mensagem de erro
        perror("fork");
        exit(EXIT_FAILURE);
    }
}

void executeCommandWithOutputRedirection(char *cmd, char **args, char
*outputFile) {
    pid_t pid = fork();

    if (pid == 0) {
        //Código do processo filho

        //Abre o ficheiro no modo "write"
        int fd = open(outputFile, O_WRONLY | O_CREAT | O_TRUNC, 0666);
        if (fd == -1) {
            perror("open");
            exit(EXIT_FAILURE);
        }

        //Redireciona a saida para o ficheiro
        if (dup2(fd, STDOUT_FILENO) == -1) {
            perror("dup2");
            close(fd);
            exit(EXIT_FAILURE);
        }

        //Fecha o descritor não necessario apos a duplicacao
        close(fd);

        //Executa o comando
        execvp(cmd, args);
```

```
        //Se o exec falhar, mostra uma mensagem de erro
        perror("execvp");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        //Código do processo pai
        int status;
        waitpid(pid, &status, 0);

        if (WIFEXITED(status) && WEXITSTATUS(status) != 0) {
            fprintf(stderr, "O comando '%s' falhou com o código de saída %d\n",
cmd, WEXITSTATUS(status));
            exit(EXIT_FAILURE);
        }
    } else {
        //Se o fork falhar, mostra uma mensagem de erro
        perror("fork");
        exit(EXIT_FAILURE);
    }
}

void executeCommandWithPiping(char *cmd1, char **args1, char *cmd2, char
**args2) {
    int pipefd[2];
    pid_t pid1, pid2;

    if (pipe(pipefd) == -1) {
        //Se o pipe falhar, mostra uma mensagem de erro
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    pid1 = fork();
    if (pid1 == 0) {
        //Código do processo filho do comando 1

        //Fecha o "read" desnecessario no fim do "pipe"
        close(pipefd[0]);
        //Redireciona o "stdout" para o "write" no fim do "pipe"
        dup2(pipefd[1], STDOUT_FILENO);
        //Fecha o "write" no fim do "pipe"
        close(pipefd[1]);

        //Executa o comando 1
        execvp(cmd1, args1);

        //Se o exec do comando 1 falhar, mostra uma mensagem de erro
        perror("execvp cmd1");
        exit(EXIT_FAILURE);
    } else if (pid1 < 0) {
        //Se o fork do comando 1 falhar, mostra uma mensagem de erro
        perror("fork cmd1");
        exit(EXIT_FAILURE);
    }

    pid2 = fork();
    if (pid2 == 0) {
        //Código do processo filho do comando 2

        //Fecha o "write" desnecessario no fim do "pipe"
        close(pipefd[1]);
        //Redireciona o "stdin" para o "read" no fim do "pipe"
        dup2(pipefd[0], STDIN_FILENO);
        //Fecha o "read" no fim do "pipe"
        close(pipefd[0]);
```

```
        //Executa o comando 2
        execvp(cmd2, args2);

        //Se o exec do comando 2 falhar, imprime uma mensagem de erro
        perror("execvp cmd2");
        exit(EXIT_FAILURE);
    } else if (pid2 < 0) {
        //Se o fork do comando 2 falhar, imprime uma mensagem de erro
        perror("fork cmd2");
        exit(EXIT_FAILURE);
    }

    //Código do processo pai
    close(pipefd[0]);
    close(pipefd[1]);
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
}

int main(int argc, char *argv[]) {
    //Variavel relativa ao numero de argumentos sem contar com "mycmd"
    int args = argc - 1;

    //Verifica se só existe 1 argumento e se este é "top"
    if (args == 1 && strcmp(argv[1], "top") == 0) {
        executeCommandTop();
        return 0;
    }

    //Variáveis relativas aos nomes dos ficheiros, presenca de operadores,
operador e argumentos do comando
    char *inputFile = NULL;
    char *outputFile = NULL;
    int found = 0;
    char *operator = NULL;
    int argc_cmd1 = 0;
    int argc_cmd2 = 0;

    //Aloca memoria para os arrays de argumentos do comando
    char **argv_cmd1 = malloc((argc + 1) * sizeof(char *));
    char **argv_cmd2 = malloc((argc + 1) * sizeof(char *));

    //Verifica se a alocacao de memoria foi bem-sucedida
    if (argv_cmd1 == NULL || argv_cmd2 == NULL) {
        fprintf(stderr, "Erro ao alocar memória.\n");
        exit(EXIT_FAILURE);
    }

    //Loop para percorrer os argumentos
    for (int i = 1; i < argc; i++) {
        //Verifica se o argumento é um operador
        if (strcmp(argv[i], ">") == 0 || strcmp(argv[i], "<") == 0 ||
strcmp(argv[i], "|") == 0) {
            found = 1;
            operator = argv[i];
            continue;
        }

        //Se nao for encontrado nenhum operador, adiciona ao array cmd1
        if (!found) {
            argv_cmd1[argc_cmd1++] = strdup(argv[i]);
        } else {
            //Se foi encontrado um operador, adiciona ao array cmd2
```

```
            argv_cmd2[argc_cmd2++] = strdup(argv[i]);
        }
    }

    //Adiciona NULL no fim dos arrays para indicar o fim
    argv_cmd1[argc_cmd1] = NULL;
    argv_cmd2[argc_cmd2] = NULL;

    //Atraves da variavel "found" ve se foi encontrado algum argumento
    if (found) {
        if (strcmp(operator, "<") == 0) {
            //Operador referente ao input
            if (argc_cmd2 > 0) {
                inputFile = argv_cmd2[0];
            }
            executeCommandWithInputRedirection(argv_cmd1[0], argv_cmd1,
inputFile);
        } else if (strcmp(operator, ">") == 0) {
            //Operador referente ao output
            if (argc_cmd2 > 0) {
                outputFile = argv_cmd2[0];
            }
            executeCommandWithOutputRedirection(argv_cmd1[0], argv_cmd1,
outputFile);
        } else if (strcmp(operator, "|") == 0) {
            //Operador referente ao pipe
            executeCommandWithPiping(argv_cmd1[0], argv_cmd1, argv_cmd2[0],
argv_cmd2);
        }
    } else {
        //Caso nao tenha sido encontrado nenhum operador, executa o comando de
forma normal
        executeCommand(argv_cmd1[0], argv_cmd1);
    }

    //Liberta a memoria alocada anteriormente para evitar "memory leaks"
    for (int i = 0; i < argc_cmd1; i++) {
        free(argv_cmd1[i]);
    }
    for (int i = 0; i < argc_cmd2; i++) {
        free(argv_cmd2[i]);
    }
    free(argv_cmd1);
    free(argv_cmd2);

    return 0;
}
```