

# RELATÓRIO DE SISTEMAS OPERATIVOS

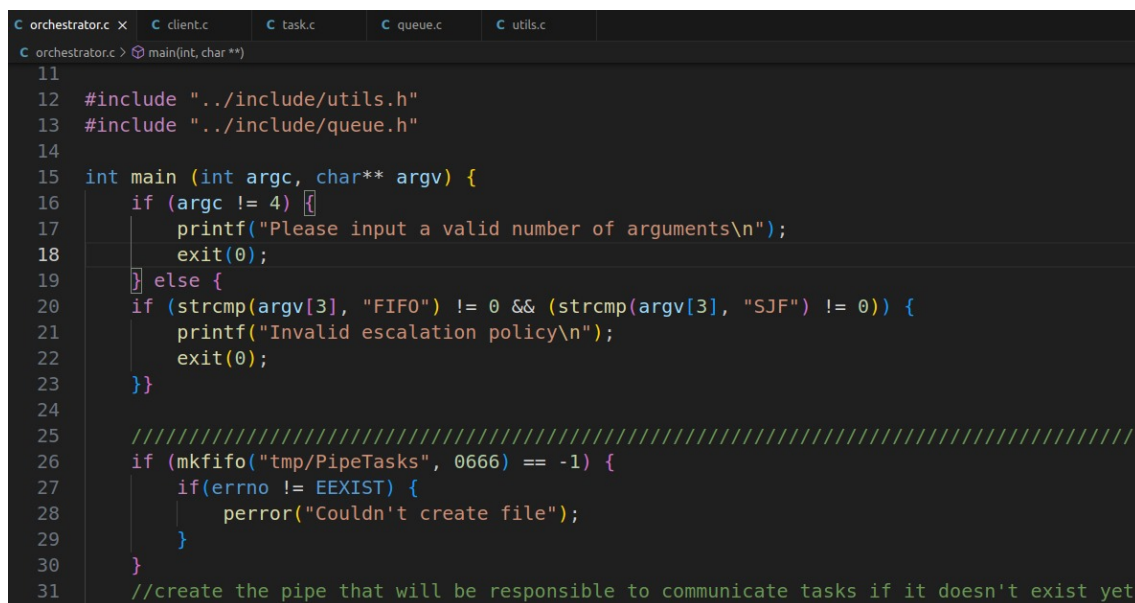
## GRUPO 4:

- André Filipe Pereira Ribeiro, A104436
- Jorge Rafael Machado Fernandes, A104168
- Pedro Miguel Araújo Gomes, A104540

## COMUNICAÇÃO ENTRE CLIENTE E SERVIDOR

A comunicação entre clientes e servidor foi o primeiro passo e, dada a sua importância, procurámos garantir que tínhamos o controlo total desta comunicação, de forma a não comprometer o projeto em nenhuma das fases posteriores.

### Ler tarefas



```
11
12 #include "../include/utils.h"
13 #include "../include/queue.h"
14
15 int main (int argc, char** argv) {
16     if (argc != 4) {
17         printf("Please input a valid number of arguments\n");
18         exit(0);
19     } else {
20         if (strcmp(argv[3], "FIFO") != 0 && (strcmp(argv[3], "SJF") != 0)) {
21             printf("Invalid escalation policy\n");
22             exit(0);
23         }
24     }
25
26     //////////////////////////////////////
27     if (mkfifo("tmp/PipeTasks", 0666) == -1) {
28         if(errno != EEXIST) {
29             perror("Couldn't create file");
30         }
31     }
32     //create the pipe that will be responsible to communicate tasks if it doesn't exist yet
```

Ao iniciar o programa, o nosso servidor cria um pipe FIFO, “PipeTasks”, sendo este o canal de comunicação pelo qual optámos, de forma a tornar possível a receção de tarefas. Seja sobre a forma de pedidos de execução ou pedidos de status, provenientes do cliente ou até tarefas já terminadas vindas de processos filho criados pelo próprio servidor.

### O QUE É UMA TAREFA?

Tal como foi referido anteriormente existem 3 tipos de tarefas. Desses 3 tipos conseguimos ainda dividi-las em 2 grupos.

- Grupo Cliente (tarefas provenientes do Cliente) – Pedidos de execução ou pedidos de status.
- Grupo Servidor (tarefas provenientes de processos filhos criados pelo próprio servidor) – Pedidos de execução já terminados.

Dados os atributos da nossa estrutura “Task” distinguimos as diferentes tarefas da seguinte maneira:

```
typedef struct {
    int inputed_time_ms;
    char flag[3];
    char toExecute[278];
    int status;
    int id;
    int done;
} Task;
```

int inputed\_time\_ms - Guarda o valor do tempo previsto dado para um pedido de execução, (pedidos de execução já terminados vão herdar esse tempo). Para pedidos de status este tempo fica com o valor default igual a 0;

char flag[3] – Este atributo distingue através das flags “-u” e “-p”, respetivamente, pedidos de execução de programas singulares, de pedidos de execução do número de programas variável.

char toExecute[278] – Este atributo guarda a mensagem que contém o(s) programa(s) e respetivos argumentos (no caso da existência de argumentos) para qualquer pedido de execução. O seu tamanho foi escolhido dado a condição de input de argumentos  $\leq 300$  bytes.

int status – Para pedidos de status, este atributo assume o valor 1 e, para pedidos de execução assume o valor 0.

int id – Este atributo guardará o id de cada pedido de execução, logo, será apenas preenchido (isto é, com um valor diferente do default), depois do servidor atribuir um id a esse mesmo pedido de execução.

int done – Este atributo irá distinguir pedidos de execução de grupo cliente de pedidos de execução de grupo servidor. O valor deste atributo para todos os pedidos de execução do grupo cliente será 0, e do grupo servidor será 1, visto que este campo só será alterado no fim da execução dessa determinada tarefa.

## IDs de tarefa e status

De forma a comunicar com o utilizador, os nossos clientes precisam de receber IDs de tarefa gerados e provenientes do servidor e mensagens de status também provenientes de servidor. Para este efeito, criámos mais 2 pipes FIFO, desta vez no cliente, pois como temos N clientes e apenas 1 servidor, cada cliente cria os seus FIFOs momentaneamente.

```

13 int main (int argc, char** argv) {
14     // ...
15 }
16
17 else if (strcasecmp(argv[1], "Execute") == 0) { //request to server to execute a task
18
19     if (argc < 5) {
20         printf("Please, input a valid number of arguments to request the execution of a task");
21     } else {
22         //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
23         if (mkfifo("tmp/PipeIDs", 0666) == -1) {
24             if(errno != EEXIST) {
25                 perror("Couldn't create file");
26             }
27         }
28         //create the pipe that will be responsible to communicate Tasks' IDs if it doesn't exist yet
29
30         Task tarefa; //load a task
31         setTaskFlag(argv[3], &tarefa);
32         setTaskTime(atoi(argv[2]), &tarefa);
33         setTaskToExecute(argv[4], &tarefa);
34         setTaskStatus(0, &tarefa); //declare that this task isn't a status request
35         setTaskDone(&tarefa); //by default, this will set the "done" to 0, meaning this is a task that came from the client
36
37         int fd = open("tmp/PipeTasks", O_WRONLY); //open pipe to write task
38         write(fd, &tarefa, sizeof(tarefa)); //write task to fifo pipe
39         close(fd); //close pipe descriptor
40
41         int id;
42         int fd2 = open("tmp/PipeIDs", O_RDONLY); //open pipe to read the task id
43         read(fd2, &id, sizeof(int)); //stores the task id into the variable "id"
44         close(fd2); //close pipe descriptor
45         printf("O ID da sua tarefa é %d\n", id);
46
47         // Unlink (destroy) the FIFO pipe
48         if (unlink("tmp/PipeIDs") == -1) {
49             perror("Error unlinking FIFO pipe");
50         }
51     }
52 }

```

## Políticas de escalonamento

O nosso projeto foi concebido para praticar 2 políticas de escalonamento. Política FIFO, First In First Out, que funciona por uma fila simples onde o primeiro pedido de execução a ser enviado será o primeiro a executar. E política SJF, Shortest Job First, que tira partido do tempo de execução estimado para um dado pedido de execução. Como o nome indica, irá ser executado o pedido de execução que tiver o menor tempo estimado.

```

int main (int argc, char** argv) {
    if (argc != 4) {
        printf("Please input a valid number of arguments\n");
        exit(0);
    } else {
        if (strcmp(argv[3], "FIFO") != 0 && (strcmp(argv[3], "SJF") != 0)) {
            printf("Invalid escalation policy\n");
            exit(0);
        }
    }
}

```

## Inicialização do Servidor

Antes de entrar no ciclo infinito que manterá o nosso servidor sempre aberto, inicializámos a seguintes variáveis que são cruciais ao bom funcionamento do projeto.

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
int idGenerator = 0; //variable that will fit as an Unique ID Generator for each Task that comes from a Client
int parallelTasks = atoi(argv[2]); //number of tasks that this Server can execute simultaneously
int tasksInExecution = 0; //variable to monitor how many tasks are being executed simultaneously
struct Queue* taskInQueue = createQueue(20); //data struct to store Tasks that are on waitlist to be executed
struct Queue* InExecution = createQueue(parallelTasks); //data struct to store Tasks that are being executed
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

Neste tópico refugiamos a nossa explicação na nossa documentação de forma a evitar redundância.

## Tratamento de tarefa

```
////////////////////////////////////
while (1) { //infinite cycle that keeps the Server running
    Task tarefa; //buffer task
    int fdtarefa = open("tmp/PipeTasks", O_RDONLY); //most important pipe, responsible for handling tasks

    while (read(fdtarefa, &tarefa, sizeof(tarefa)) > 0) {

        if (getTaskStatus(&tarefa) == 0) { //if it ain't a status request

            if (getTaskIsDone(&tarefa)) { //Task that ended its execution
                removeTaskById(InExecution, getTaskId(&tarefa)); //remove the task from the Tasks in execution data structure
                tasksInExecution--; //decrement number of tasks executing at the present time
            }

            if (getTaskIsDone(&tarefa) == 0) { // Task that came from Client
                idGenerator++; //increment taskID
                int fdtarefa2 = open("tmp/PipeIDs", O_WRONLY); //open pipe to "send" the taskID to the Client
                write(fdtarefa2, &idGenerator, sizeof(int));
                close(fdtarefa2); //close descriptor
                setTaskId(idGenerator, &tarefa); //"attach" the taskID to the task
                enqueue(taskInQueue, tarefa); //store the task in the waitlist
            }
        }
    }
}
```

Esta é a parte do código, anterior à execução de tarefas e criação de processos filho, onde passam todos os tipos de tarefa. Depois de abrir o ciclo infinito que mantém o servidor a “rodar”, inicializamos o descritor de abertura do pipe que irá ler tarefas. Assim, inicializamos o ciclo de leitura que estará sempre pronto para ler se existir alguém que escreva no pipe do outro lado. A cada tarefa recebida todas têm de ser distinguidas pelas diferentes condições que tiram informação dos atributos da estrutura “Task”. Depois de distinguidas entre os 3 tipos de tarefa, todas iram receber um tratamento diferente como é possível perceber pelo código e a sua documentação.

## Começo da execução

Depois de entendermos com que tipo de tarefa estamos a lidar, abordamos os pedidos de execução que estão à espera de ser executados caso haja pedidos a executar e caso haja “espaço” para os executar, isto é, o número de “tasks” em execução seja menor que o número de “tasks” que podem ser executadas em paralelo através do argumento dado no input ao inicializar o servidor. De forma a perceber que tarefa vamos executar fazemos recurso à nossa política. A partir do momento que os nossos algoritmos determinam que tarefa vamos executar, removemos a tarefa da lista de espera e adicionamo-la à lista de tarefas em execução.

Como sabemos o tipo de execução que vamos realizar será determinado pela flag da tarefa a executar:

```
65 if the number of tasks being executed is smaller than the fixed number of tasks that can be executed in parallel and if the waitlist ain't empty
66 if (tasksInExecution < parallelTasks && !isEmpty(taskInQueue)) {
67     Task firstInQueue;
68     if (strcmp(argv[3], "FIFO") == 0) {
69         firstInQueue = front(taskInQueue); //get a copy of the first task of the queue
70     } else if (strcmp(argv[3], "SJF") == 0) {
71         firstInQueue = getMinInputedTimeTask(taskInQueue);
72     }
73
74     removeTaskById(taskInQueue, getTaskId(&firstInQueue)); //remove the task from the waitlist since is gonna be executed
75     enqueue(InExecution, firstInQueue); //store the task in the executing queue
76
77
78     //if the flag given with the task was "-u"
79     if (strcmp(getTaskFlag(&firstInQueue), "-u") == 0) {
80         if (mkdir(argv[1], 0777) == -1) {
81             // Check if the error is because the directory already exists
82             if (errno != EEXIST) {
83                 perror("Couldn't create directory");
84             }
85         }
86     }
87 }
```

## Execução

### Flag “-u”

Para executar pedidos de execução com flag “-u” tivemos de “chamar” um “fork()” que nos permita tornar independente a execução de cada tarefa, deixando o servidor continuar a iterar à espera de novas “tasks”. Posto isto, criámos outro “fork()” que será o verdadeiro responsável por executar a tarefa. Tirámos o “id” da tarefa para criar o seu ficheiro individual e redirecionámos toda a informação tanto do System.Out que corresponde ao valor 1 como do System.err que corresponde ao valor 0 para esse mesmo ficheiro. Através do comando “execvp” e já com todos os argumentos da tarefa devidamente separados, executamos a tarefa.

```
//fork that is gonna separate the execution part of the server and makes sure that the server keeps being able to receive tasks
if (fork() == 0) {
    char *args[20]; // Assuming a maximum of 20 arguments
    splitString(getTasktoExecute(&firstInQueue), " ", args); //Split the task arguments to execute
    struct timeval start, end;
    gettimeofday(&start, NULL); //start counting time of execution of task

    //fork that is gonna actually execute the task
    if (fork() == 0) {
        char fileoutput[20]; //file name buffer
        snprintf(fileoutput, sizeof(fileoutput), "%s/Task%d.txt", argv[1], getTaskId(&firstInQueue)); //init the output file name
        int fdFO = open(fileoutput, O_CREAT | O_APPEND | O_WRONLY, 0666); //create the output file
        dup2(fdFO, 1); //redirect system.out executions to Task"id".txt
        dup2(fdFO, 2); //redirect system.err

        char *TaskID = createTaskString(getTaskId(&firstInQueue)); //header for the output file
        write(1, TaskID, strlen(TaskID)); //write header
        free(TaskID); //free memory

        execvp(args[0], args); //execute the program and its arguments + write execution output in the output file
        perror("exec failed\n"); //if it reaches this line it means there was an error, because the exec should have killed the process
        _exit(1);
    } else {
        char fileoutput[20];
        snprintf(fileoutput, sizeof(fileoutput), "%s/Task%d.txt", argv[1], getTaskId(&firstInQueue)); //init the output file name
        int fdFO = open(fileoutput, O_CREAT | O_APPEND | O_WRONLY, 0666);

        wait(NULL); //wait for the child process
        gettimeofday(&end, NULL); //stop counting the time after waiting for the child process to "die"
        int timeExec = (end.tv_sec - start.tv_sec) * 1000 + (end.tv_usec - start.tv_usec) / 1000;
        char* time = createTaskTime(timeExec); //create baseboard for output file
        write(fdFO, time, strlen(time)); //write baseboard for output file
        free(time); //free time
        close(fdFO); //close file descriptor
        TaskisDone(&firstInQueue); //set done attribute to done = 1
        int fdtaskaw = open("tmp/PipeTasks", O_WRONLY); //init pipe descriptor
        write(fdtaskaw, &firstInQueue, sizeof(firstInQueue)); // write in "PipeTasks"
        close(fdtaskaw); //close pipe descriptor

        char* tasklinedone = createTaskLineDone(getTaskId(&firstInQueue), getTasktoExecute(&firstInQueue), timeExec); //create line for a done task
        int fdTasksDone = open("output/tasksDone.txt", O_WRONLY | O_CREAT | O_APPEND, 0666); //open the file descriptor to store all the done tasks
        write(fdTasksDone, tasklinedone, strlen(tasklinedone)); //write the line in the done tasks file
        close(fdTasksDone); //close file descriptor
        _exit(0);
    }
}
_exit(0);
}
```

O término da execução da tarefa é marcado pela espera do pai do processo que a executou. Com o fim dessa espera, vem o fim da contagem do tempo e vem a escrita dessa tarefa, agora como um pedido de execução terminado, no pipe principal do servidor que está constantemente à espera de tarefas para ler.

Para além disso, a informação remetente a esta tarefa é ainda guardada no ficheiro das tarefas acabadas. Ficheiro esse que contém as informações precisas para responder a pedidos de status.

## Flag “-p”

A execução para a flag “-p” é muito semelhante à execução para a flag “-u”, simplesmente como estamos a executar vários programas e queremos “misturar” os seus resultados faremos 2 adições importantes. Primeiro criámos um ciclo “for” dentro do primeiro “fork()” que irá iterar 1 vez por programa, ou seja, irá fazer 1 “fork()” de execução de programa por cada programa. Depois de forma a “misturar” os programas iremos redirecionar sempre o output do programa de ordem “x” para um pipe sem nome e redirecionar o stdin do programa de ordem “x+1” para que leia do pipe. O primeiro programa não precisa de ter o seu stdin

redirecionado assim como o último programa não precisa de ter o seu stdout redirecionado pois irá escrever no ficheiro de output como acontece com a flag “-u”.

```
135         } else { //if the flag given with the task was "-p"
136             if (mkdir(argv[1], 0777) == -1) {
137                 // Check if the error is because the directory already exists
138                 if (errno != EEXIST) {
139                     perror("couldn't create directory");
140                 }
141             }
142         }
143
144         char *commands[20]; //assuming a max of 20 programs/arguments
145         int occupancy = fillArray(commands, getTasktoExecute(&firstInqueue)); //separating the programs but not its arguments if they exist
146                                     //and find out the number of programs we're gonna execute together
147         if (fork() == 0) {
148             int fdIN = 0; //pipe descriptor thats gonna connect the stdin of a program with the stdout of the last program
149             int timeExec; //variable to store the time of the task
150             char* time; //buffer for the baseboard
151             struct timeval start, end;
152             gettimeofday(&start, NULL); //start counting time of execution of task
153             for(int i=0; i<occupancy; i++) { //1 iteration per program
154                 int fd[2]; //nameless pipe to connect the stdouts and stdins of prgrams
155                 pipe(fd);
156                 if (fork() == 0) {
157                     // if it ain't the first program*/ if (i!=0) dup2(fdIN, 0); //define stdin for this process
158                     // if it ain't the last program*/ if (i < occupancy - 1) {
159                         dup2(fd[1], 1); //define stdout for this process
160                     }
161                     close(fd[0]); // Closes the reading side of the pipe
162                     if (i == occupancy-1) { //if it is the last program it will write the output in the output file
163                         char fileoutput[20];
164                         snprintf(fileoutput, sizeof(fileoutput), "%s/Task%d.txt", argv[1], getTaskId(&firstInqueue));
165                         int fdFO = open(fileoutput, O_CREAT | O_APPEND | O_WRONLY, 0666);
166                         dup2(fdFO, 1); //redirect system.out executions to Task"i".txt
167                         dup2(fdFO, 2); //redirect system.err
168
169                         char *TaskID = createTaskString(getTaskId(&firstInqueue));
170                         write(1, TaskID, strlen(TaskID));
171                         free(TaskID);
172                     }
173                     exec_command(commands[i]); //execute and parse a program if there exist other arguments + writes in the pipe
174                     perror("exec failed\n");
175                     exit(1);
176                 }
177                 wait(NULL); //wait for child process
178                 close(fd[1]); //close the writing side of the pipe
179                 fdIN = fd[0]; //next program will receive the output for his input
180                 if (i == occupancy - 1) {
181                     gettimeofday(&end, NULL); //finish counting
182                     timeExec = (end.tv_sec - start.tv_sec) * 1000 + (end.tv_usec - start.tv_usec) / 1000;
183                     time = createTaskTime(timeExec);
184                     char fileoutput[20];
185                     snprintf(fileoutput, sizeof(fileoutput), "%s/Task%d.txt", argv[1], getTaskId(&firstInqueue));
186                     int fdFO = open(fileoutput, O_CREAT | O_APPEND | O_WRONLY, 0666);
187                     write(fdFO, time, strlen(time));
188                     free(time);
189                     close(fdFO);
190                     TaskisDone(&firstInqueue); //set attribute done to done = 1
191                     int fdtaskDone = open("tmp/PipeTasks", O_WRONLY);
192                     write(fdtaskDone, &firstInqueue, sizeof(firstInqueue));
193                     close(fdtaskDone);
194
195                     char* tasklinedone = createTaskLineDone(getTaskId(&firstInqueue), getTasktoExecute(&firstInqueue), timeExec);
196                     int fdTasksDone = open("output/tasksDone.txt", O_WRONLY | O_CREAT | O_APPEND, 0666);
197                     write(fdTasksDone, tasklinedone, strlen(tasklinedone));
198                     close(fdTasksDone);
199                 }
200             }
201             _exit(0);
202         }
203     }
204     tasksInExecution++; //increment tasks in execution
205 }
```

## Pedido de Status

Ao receber um pedido de status tivemos uma complexidade de trabalho não tão elevada. Apenas criámos 3 strings que foram depois concatenadas em 1 string que é a nossa mensagem de status a enviar para o Cliente que passará para o utilizador. Dessas 3 strings:

- 1 string com a informação presente na estrutura de dados de tarefas em espera;
- 1 string com a informação presente na estrutura de dados de tarefas em execução;
- 1 string com a informação presente no ficheiro que contém as tarefas concluídas.

```

207     } else { //is a status request
208         char* inQueue = createQueueStatusInQueue(taskInQueue); //string with all the info from the tasks waitlist data structure
209         char* executing = createQueueStatusExecuting(InExecution); //string with all the info from the data structure of the executing tasks
210
211         int fdTasksDone = open("output/tasksDone.txt", O_RDONLY); //Open in read-only mode
212         if (fdTasksDone == -1) {
213             perror("Error opening file");
214         }
215
216         off_t tamanho = lseek(fdTasksDone, 0, SEEK_END); //size of the file that stores all the done tasks
217         lseek(fdTasksDone, 0, SEEK_SET); // Reset file pointer to the beginning
218         char* statusDone = malloc(tamanho + 1 + 15);
219         strcpy(statusDone, "--Done--\n\n"); // Initialize statusDone with "--Done--\n\n"
220         char* linesDone = malloc(tamanho + 1); // Allocate memory
221
222         if (linesDone == NULL) {
223             perror("Memory allocation failed");
224         }
225
226         // Read content from the file into linesDone if the file is not empty
227         ssize_t bytes_read = 0;
228         if (tamanho > 0) {
229             bytes_read = read(fdTasksDone, linesDone, tamanho);
230             if (bytes_read == -1) {
231                 perror("Error reading file");
232                 exit(EXIT_FAILURE);
233             }
234         }
235
236         // Null-terminate linesDone if bytes were read
237         if (bytes_read > 0) {
238             linesDone[bytes_read] = '\0';
239         }
240
241         strcat(statusDone, linesDone); // Concatenate linesDone to statusDone
242         statusDone[strlen(statusDone)] = '\0'; // Ensure statusDone is properly null-terminated
243         close(fdTasksDone);
244
245         char status[4500];
246         strcpy(status, inQueue); // Initialize status with inQueue
247         strcat(status, executing); // Concatenate executing to status
248         strcat(status, statusDone); // Concatenate statusDone to status
249
250
251         int fdStatus = open("tmp/PipeStatus", O_WRONLY);
252         write(fdStatus, status, strlen(status)); //write status message thats gonna be read at the other side of the pipe by the client
253         close(fdStatus);

```