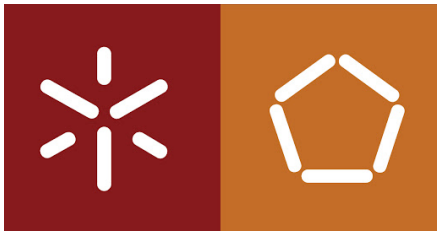


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Sistemas Operativos

Trabalho Prático

Grupo 39

7 de maio de 2024

Desenvolvido por:

Gonçalo Alves - a104079
João Cunha - a104611
João Sá - a104612

Conteúdo

1 Introdução

O objetivo de deste relatório é apresentar o desenvolvimento do trabalho prático da UC Sistemas Operativos no ano letivo 2023/2024.

Este trabalho consiste na implementação de um orquestrador de tarefas através da conexão entre "cliente-servidor". Utilizando o orquestrador, os utilizadores podem submeter tarefas para execução, especificando a sua duração e o comando a ser executado.

2 Descrição do Sistema

Tal como referido anteriormente, o programa consiste num sistema de orquestração de tarefas.

Para a utilização do sistema, um utilizador deve especificar ao programa como é que o mesmo deve proceder à execução da tarefa que deseja realizar (um ou vários programas), e a duração estimada para a execução dos mesmos.

Portanto, tal como podemos concluir haverá dois comandos para a execução das tarefas que o utilizador desejar.

Execução de uma tarefa com apenas um programa

```
./client execute time -u "prog-name [args]"

Exemplo de utilizacao:
./client execute 10 -u "sleep 10"
```

Execução de uma tarefa com um conjunto de programas

```
./client execute time -p "prog-a [args] | prog-b [args] | prog-c [args]"

Exemplo de utilizacao:
./client execute 100 -u "cat fich1 | grep "palavra" | wc -l"
```

O utilizador, pode também, fazer um inquérito ao servidor para obter um registo do estado programa (i.e., saber quais são as tarefas que estão em fila de espera, as que estão a ser executadas e também as que já foram executadas) através do comando:

```
./client status
```

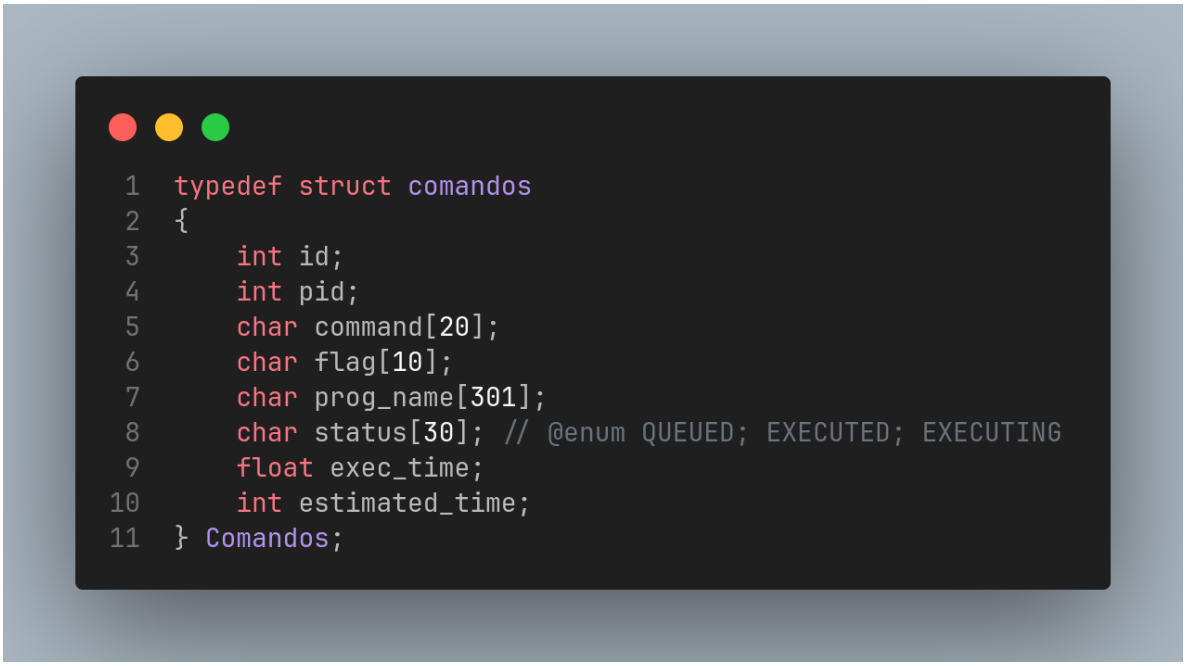
Por fim, o utilizador termina a sessão, ou seja desliga o servidor através do comando:

```
./client shutdown
```

3 Abordagem

3.1 Parsing

Para guardar as informações necessárias para a execução correta da tarefa e das possíveis consultas do utilizador (tais como o comando status), decidimos criar uma struct *Comandos*.



```c
typedef struct comandos
{
    int id;
    int pid;
    char command[20];
    char flag[10];
    char prog_name[301];
    char status[30]; // @enum QUEUED; EXECUTED; EXECUTING
    float exec_time;
    int estimated_time;
} Comandos;
```

Figura 1: Estrutura responsável por armazenar os dados de uma Tarefa

- O inteiro *ID* armazena o ID de determinada tarefa
- O inteiro *PID* armazena o pid (process ID) associado à tarefa quando a mesma é executada.
- A string *COMMAND* armazena qual é o tipo de execução que o sistema vai realizar (e.g. status, execute, shutdown).
- A string *FLAG* armazena a flag que indica o tipo de execução (-u"ou -p").
- A string *PROG NAME* armazena os comandos que vão ser executados (incluindo argumentos).
- A string *STATUS* armazena o estado de uma tarefa (QUEUED / EXECUTED / EXECUTING).
- O float *EXEC TIME* armazena o tempo que a tarefa demorou a ser executada.
- O inteiro *ESTIMATED TIME* armazena o tempo que se estima que a tarefa irá demorar a ser executada.

Esta estrutura está responsável por armazenar qualquer tipo de instrução do utilizador (i.e., instruções como status e shutdown também serão representadas nesta estrutura, mas os valores como flag são ignorados).
Após o preenchimento adequado da estrutura, a mesma será escrita no fifo do servidor. Posteriormente a este processo, o servidor procederá à leitura do fifo, armazenando a tarefa. O campo *COMMAND* será comparado com os casos possíveis, para proceder à execução da tarefa desejada.

3.2 Modularidade

De modo a facilitar a manutenção, a legibilidade e a divisão do código optámos por criar quatro módulos (*Client, Orchestrator, Queue, Commands*). Esta técnica garante que implementações futura continuem concisas com o projeto já desenvolvido devido à sua organização.

3.2.1 Client

Este módulo, tal como o nome indica, é responsável pelo cliente. Isto é, está responsável para primeira comunicação entre o cliente e o servidor. Para isso, recebe o input do utilizador e consoante o comando introduzido irá construir a struct com os dados necessários para sua execução. Após a sua construção, a estrutura é passada para o servidor, recorrendo a um FIFO, onde será realizada a sua execução.

Para realizar a comunicação entre o servidor e o cliente, decidimos implementar um FIFO para cada um de modo a aumentar a eficiência e simplicidade deste processo.

```c
else if (strcmp(argv[1], "execute") == 0)
{
    comando.estimated_time = atoi(argv[2]);
    strcpy(comando.prog_name, argv[4]);
    strcpy(comando.flag, argv[3]);

    if (strcmp(argv[3], "-u") == 0 || strcmp(argv[3], "-p") == 0)
    {
        ssize_t bytes_escritos = write(fifo_servidor, &comando, sizeof(Comandos));
        if (bytes_escritos <= 0)
        {
            write(2, "Erro ao escrever no FIFO\n", 25);
            exit(EXIT_FAILURE);
        }
    }
    int fifo_cliente = open(CLIENTE, O_RDONLY);

    int id = 0;
    ssize_t bytes_lidos = read(fifo_cliente, &id, sizeof(int));
    if (bytes_lidos <= 0)
    {
        perror("Erro ao ler do fifo\n");
        exit(EXIT_FAILURE);
    }
    snprintf(buffer, sizeof(buffer), "Task %d received\n", id);
    write(1, buffer, strlen(buffer));
    close(fifo_cliente);

}
```

Figura 2: Input de um comando "execute"no módulo Client

3.2.2 Orchestrator

O módulo *Orchestrator* é o "servidor"do projeto, ou seja será o responsável na execução das tarefas introduzidos. Para isso, recebe a estrutura previamente preenchida (no módulo anterior) através do FIFO e mantém um registro de todos os comandos (podendo estar em fila de espera, a executar e já executados) num ficheiro nomeado *Commands.log* que vai sendo atualizado a cada execução.

Ao receber uma estrutura com o comando *execute*, comando central do projeto, o servidor adiciona a mesma a uma *Queue*. Se existir espaço disponível para a execução em paralelo, o comando será executado, caso contrário ficará em espera até que um lugar fique disponível.

Após a execução, o servidor altera o *status* da tarefa para *EXECUTED* e armazena o tempo de execução e por fim atualiza o arquivo de log.

```
if (strcmp(comando_lido.command, "execute") == 0)
{
    int fifo_cliente = open(CLIENTE, O_WRONLY);
    comando_lido.id = next_task_id();
    write(fifo_cliente, &comando_lido.id, sizeof(int));
    close(fifo_cliente);

    strcpy(comando_lido.status, "QUEUED");
    add_task_toQueue(queue, comando_lido);
    close(fifo_servidor);
}
```

Figura 3: Processo de identificação de uma tarefa "execute"no módulo *Orchestrator*

No processo de execução, é verificado se um comando pode ser executado (caso exista algum comando na fila de espera e se o número de comandos em execução não ultrapassa o valor limite).

```c
if (nr_comandos < tasks_parallel && !is_queue_empty(queue))
        {
            comando_exec = queueGetNextTask(queue);
            remove_task_fromQueue(queue, comando_exec);

            strcpy(comando_exec.status, "EXECUTING");
            add_task_toQueue(received, comando_exec);
            nr_comandos++;

            pid_t pid = fork();
            if (pid == 0)
            {
                if (strcmp(comando_exec.flag, "-u") == 0) {
                    comando_exec = executa_u(fifo_servidor, received, comando_exec, logs);
                } else if (strcmp(comando_exec.flag, "-p") == 0) {
                    comando_exec = executa_p(fifo_servidor, received, comando_exec, logs);
                }
                strcpy(comando_exec.status, "EXECUTED");

                fifo_servidor = open(SERVIDOR, O_WRONLY);
                write(fifo_servidor, &comando_exec, sizeof(Comandos));
                close(fifo_servidor);
                exit(EXIT_SUCCESS);
            }
            else if (pid == -1)
            {
                perror("Erro ao criar processo-filho");
                exit(EXIT_FAILURE);
            }
        }
```

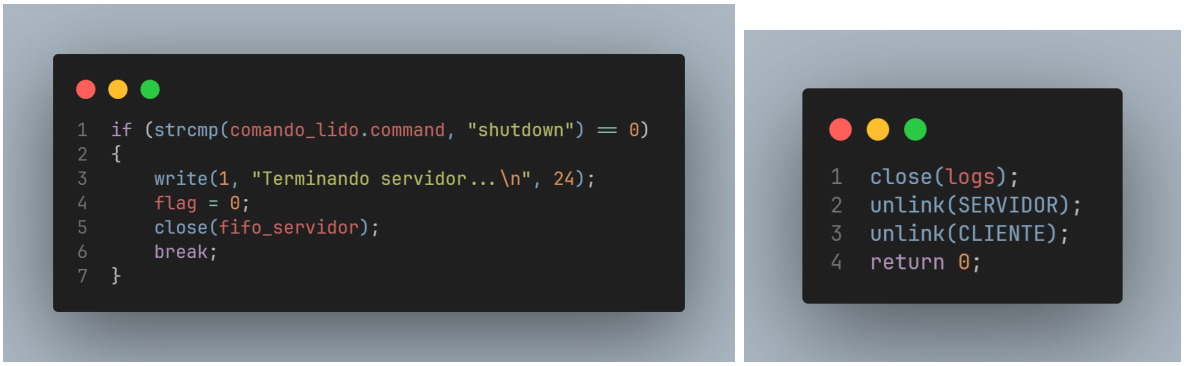
Figura 4: Processo de execução *Orchestrator*

Ao receber um comando *status*, o servidor atualizado o estado de todas as tarefas no arquivo de log e envia as informações para o cliente, recorrendo ao pipe nomeado.

```c
if (strcmp(comando_lido.command, "status") == 0)
{
    int fifo_cliente = open(CLIENTE, O_WRONLY);
    atualizaStatus(fifo_cliente, received, queue);
    close(fifo_cliente);
    close(fifo_servidor);
    continue;
}
```

Figura 5: Processo de identificação de uma tarefa "status"no módulo *Orchestrator*

Como podemos observar, o servidor continua operacional até receber uma tarefa com o comando *shutdown*. Neste momento, os fifos são limpos e o servidor cessa a sua execução.



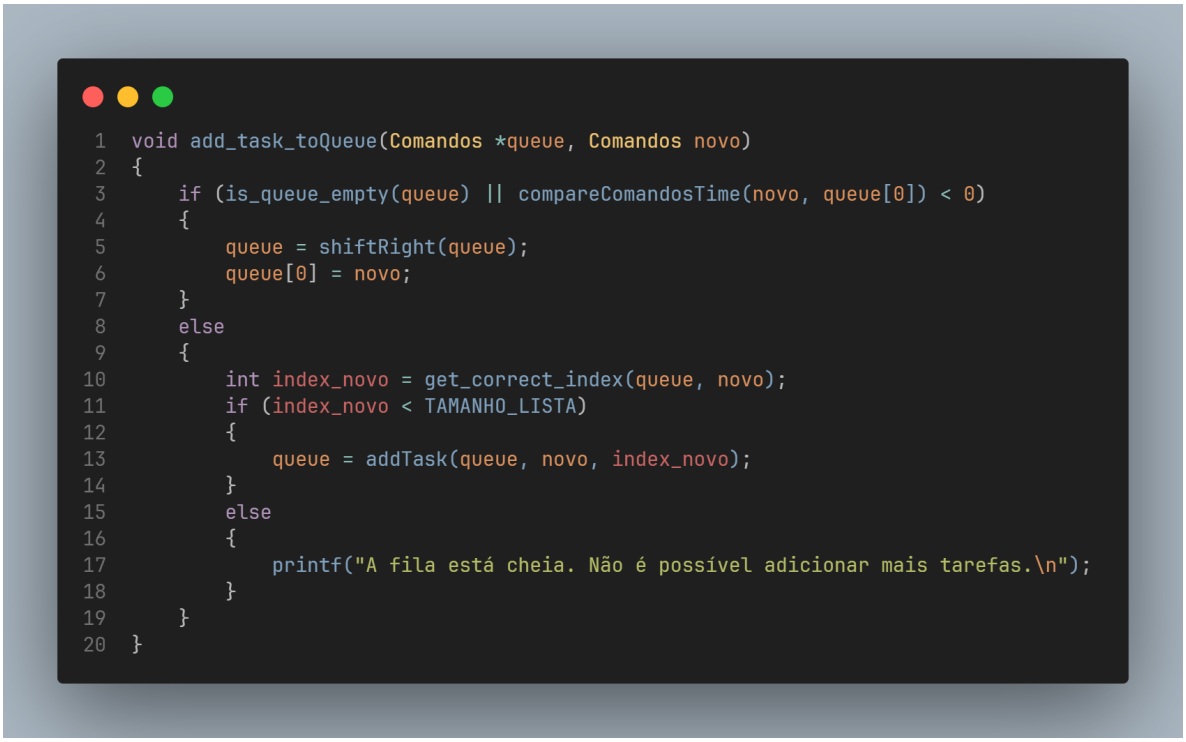
```c
if (strcmp(comando_lido.command, "shutdown") == 0)
{
    write(1, "Terminando servidor...\n", 24);
    flag = 0;
    close(fifo_servidor);
    break;
}
```

```c
close(logs);
unlink(SERVIDOR);
unlink(CLIENTE);
return 0;
```

Figura 6: Processo de identificação de uma tarefa "shutdown"no módulo *Orchestrator*

3.2.3 Queue

Este é um módulo auxiliar onde estão definidas vários funcionalidades relacionadas à fila de espera, tais como adicionar/remover tarefas à fila e obter a próxima tarefa. Este módulo é crucial para gerir eficientemente as tarefas à espera de execução, de modo a garantir uma execução ordenada e controlada. De modo a organizar as tarefas, utilizamos o tempo estimado de execução de cada tarefa.



```c
void add_task_toQueue(Comandos *queue, Comandos novo)
{
    if (is_queue_empty(queue) || compareComandosTime(novo, queue[0]) < 0)
    {
        queue = shiftRight(queue);
        queue[0] = novo;
    }
    else
    {
        int index_novo = get_correct_index(queue, novo);
        if (index_novo < TAMANHO_LISTA)
        {
            queue = addTask(queue, novo, index_novo);
        }
        else
        {
            printf("A fila está cheia. Não é possível adicionar mais tarefas.\n");
        }
    }
}
```

Figura 7: Processo de adição de uma tarefa à Queue no módulo *Queue*

3.2.4 Commands

Por fim, este módulo serve como auxílio, principalmente, na atualização das logs e do *status* das tarefas.

```c
int atualizaLogs(Comandos comando, int logs)
{
    char status[1024];
    status[0] = '\0';
    snprintf(status, sizeof(status), "ID: %d PID: %d;Time: %f ms;Prog: %s\n", comando.id, comando.pid, comando.exec_time, comando.prog_name);

    ssize_t bytes_escritos = write(logs, status, strlen(status));
    if (bytes_escritos <= 0)
    {
        write(2, "Erro ao escrever no FIFO\n", 25);
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

Figura 8: Processo de atualização das logs no módulo *Commands*

4 Scripts de Testes

De forma a testar o projeto e as suas funcionalidades, decidimos criar scripts de testes para automatizar o processo.

4.1 Script I

```bash
#!/bin/bash
# script.sh

# Caminho para o executável do cliente
CLIENT="./client"

# Envia várias tarefas para o servidor
$CLIENT execute 5000 -u "sleep 6" &
sleep 0.25
$CLIENT execute 3000 -u "sleep 5" &
sleep 0.25
$CLIENT execute 4000 -u "sleep 4" &
sleep 0.25
$CLIENT execute 2000 -u "sleep 2" &
sleep 0.25
$CLIENT execute 1000 -u "sleep 1" &
sleep 0.25
$CLIENT execute 1000 -u "echo 'Hello World'" &

# Aguarda todos os processos em background terminarem
wait
```

Figura 9: Script I

Este script realiza vários "sleep's"com argumentos diferentes, fazendo uma espera entre tarefas. Por fim realiza um "echo"que imprime "Hello World".

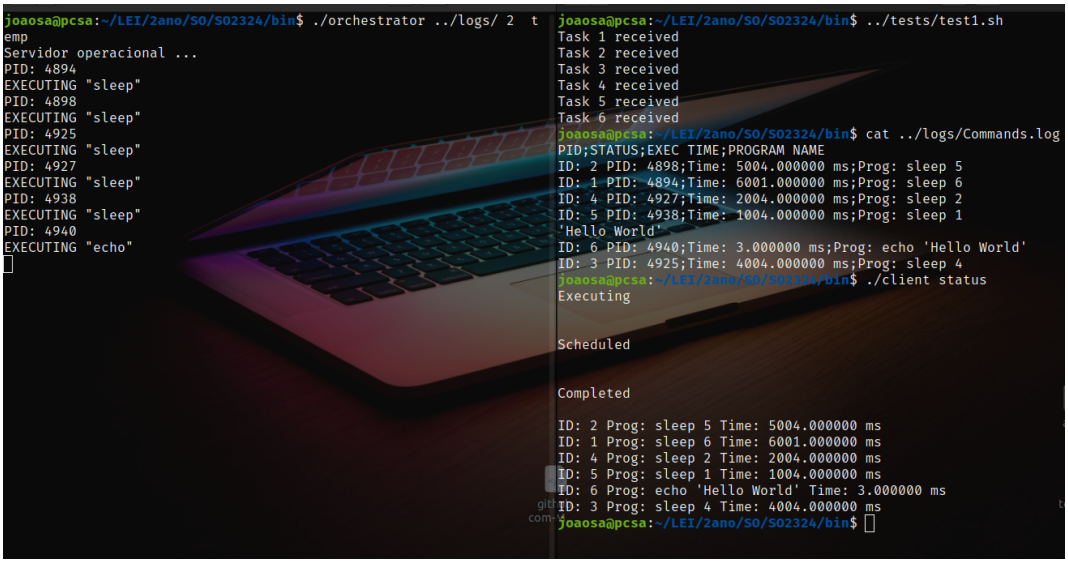


Figura 10: Output do script I

4.2 Script II

```
1  #!/bin/bash
2
3  cd ../bin
4
5  ./client execute 2 -u "ls -l" &
6  sleep 0.25
7  ./client execute 7 -u "find ." &
8  sleep 0.25
9  ./client execute 3 -u "uname -a" &
10 sleep 0.25
11 ./client execute 10 -p "cat ../src/orchestrator.c | grep "open" | wc -l"
12
13 sleep 10 &
14
15 ./client status
```

Figura 11: Script II

Tal como o outro scrip, fazemos uma espera entre tarefas mas neste caso estamos a executar tarefas mais complexas tais como “uname” e “find” para garantir que todos os comandos estão operacionais.

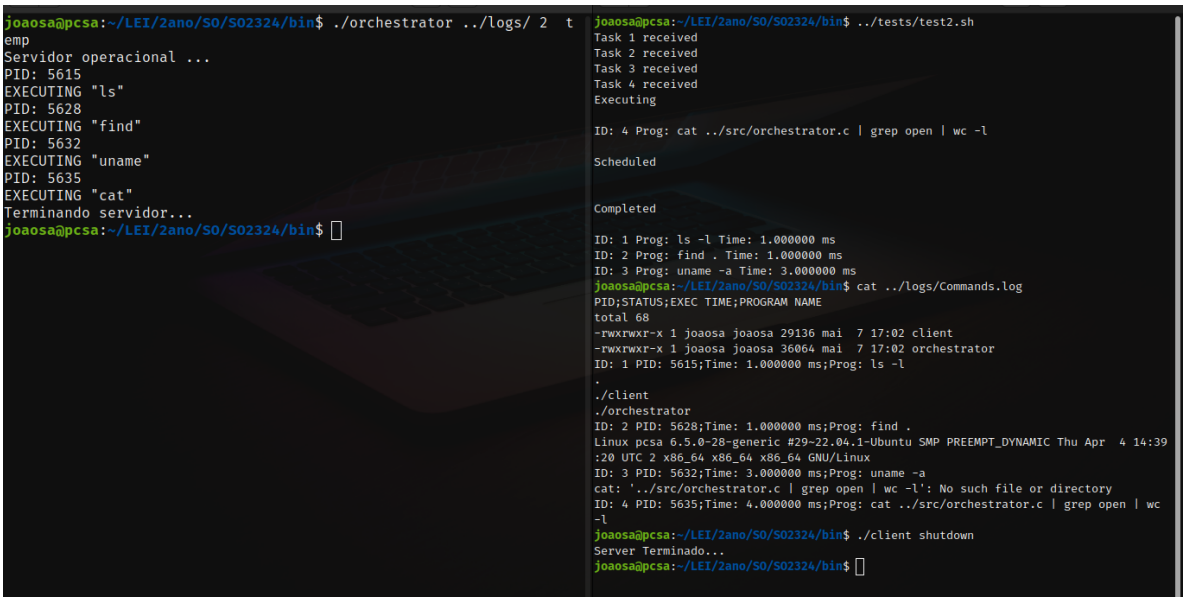


Figura 12: Output do script II

5 Decisões Tomadas

5.1 Ficheiro de Log

O nosso ficheiro de log, onde são armazenados os outputs das tarefas é apagado após o *”make clean”*, ou seja, quando decidimos reiniciar o servidor. Decidimos adotar esta abordagem pois o enunciado não especificou o método a implementar. Desta forma, cada vez que o servidor reiniciar, será criado um ficheiro de log novo. Outro método, para obter o mesmo resultado, seria apagar o ficheiro quando o servidor executar o comando *shutdown* porém, isso não iria permitar a consulta dos outputs, por parte do utilizador, após o fecho do servidor. Por este motivo, decidimos mantar a abordagem inicial.

5.2 Criação de FIFOs

Tal como referido anteriormente, decidimos criar dois FIFOs, um para o servidor e outro para o cliente. Esta abordagem, foi implementada a meio do processo de desenvolvimento pois decidimos começar com apenas um FIFO para realizar a comunicação entre o cliente e o servidor. Porém, de forma a simplificar a implementação, evitar bloqueios e aumentar a segurança (evitando corromper algum dos FIFOs) decidimos alterar a nossa abordagem inicial para a implementação de dois pipes.

5.3 Escolha de PID de tarefas encadeadas

No desenvolvimento do processo de execução de tarefas encadeadas decidimos optar por atribuir, à tarefa, o PID da execução do último comando.

6 Conclusão

De uma forma geral, o nosso trabalho possui todas as funcionalidades pedidas a funcionar. Logo, achamos que o trabalho foi bem conseguido. Tal como referido no capítulo anterior, reconhecemos que poderiamos ter optado por outros métodos (existem outros casos para além dos apresentados no capítulo 5) mas achamos que conseguimos realizar tudo de uma forma organizada e recorrendo aos conhecimentos adquiridos ao longo das aulas teóricas e práticas.