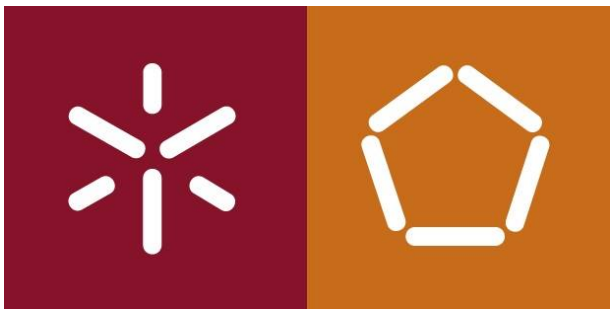

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Sistemas Operativos

Controlo e Monitorização de Processos e Comunicação

Simão Oliveira (a57041)
Rui Azevedo (a80789)
15 de Junho de 2020

Conteúdo

Lista de Figuras

1 Introdução

Este relatório tem como objectivo apresentar o desenvolvimento de um serviço de monitorização de execução e de comunicação entre processos, desenvolvido no âmbito da cadeira de **Sistemas Operativos**. O sistema foi desenvolvido na linguagem de programação ***C***, num sistema ***Unix***, usando primitivas do mesmo sistema.

O serviço desenvolvido tem como objectivo permitir a submissão de tarefas, sendo essas tarefas um conjunto de comandos encadeados por ***pipes*** **anónimos**. O sistema é capaz de monitorizar os vários processos em execução e registar o fim de cada tarefa, registando a forma como os processos acabam, que pode ser por diversas naturezas.

Numa primeira fase do relatório irá ser analisado em mais detalhe o sistema a desenvolver bem como a sua especificação. De seguida, irá ser a apresentada a arquitectura geral do sistema e como é a vista do mesmo quando se tem vários processos a correr. Por fim, irá ser apresentada, detalhadamente, a concepção de cada uma das componentes da aplicação e irão ser apresentados os testes usados para testar o programa.

2 Análise e Especificação

Nesta secção irá ser analisado o sistema a desenvolvido bem como as especificações dos requisitos para o desenvolvimento do mesmo.

2.1 Descrição do problema

O sistema desenvolvido diz respeito a um serviço de monitorização de execução de tarefas, submetidas através de um processo cliente para um processo servidor através de ***pipes* com nome**, e de comunicação entre processos.

O programa tem a capacidade de correr através da linha de comandos, indicando as opções apropriadas, e também em modo *shell*, aceitando instruções do utilizador através do *standard input*.

Existem vários comandos ao dispor do utilizador que, como foi dito anteriormente, podem ser invocados usando a linha de comandos ou a *shell* do programa. Os comandos são os seguintes:

- tempo-inactividade (-i) : definir o tempo máximo (segundos) de inactividade de comunicação num *pipe* anónimo.
- tempo-execucao (-m) : definir o tempo máximo (segundos) de execução de uma tarefa.
- executar (-e) : executar uma determinada tarefa.
- listar (-l) : listar as tarefas em execução.
- terminar (-t) : terminar uma determinada tarefa em execução.
- historico (-r) : lista o registo de todas as tarefas terminadas.
- ajuda (-h) : apresenta ajuda à utilização.

2.2 Especificação de requisitos

No desenvolvimento do programa foram usadas primitivas do sistema *Unix*, através de chamadas ao sistema, para criação de processos, criação de *pipes* anónimos e com nome, criação, leitura e escrita de ficheiros, *etc.* Foi usada também a linguagem de programação *C*, que permite executar as chamadas ao sistema, anteriormente mencionadas.

3 Arquitectura do sistema

O programa desenvolvido contém diversas componentes que cooperam entre si para tornar o sistema funcional. Os dois principais componentes do sistema são o cliente, designado por *argus*, e o servidor, designado por *argusDaemon*. O cliente envia um pedido para o servidor através de um *pipe* com nome, reservado apenas para a comunicação unidirecional *argus* → *request* → *argusDaemon*. O servidor, ao receber a mensagem, executa o pedido pretendido e, caso seja preciso dar uma resposta ao cliente, envia a resposta pelo *pipe* com nome reservado para a comunicação unidirecional *argusDaemon* → *answer* → *argus*.

A imagem abaixo apresenta a vista do sistema. As setas que contêm um ponto no início representam a relação de processo pai/filho e as setas normais representam comunicação entre *pipes*.

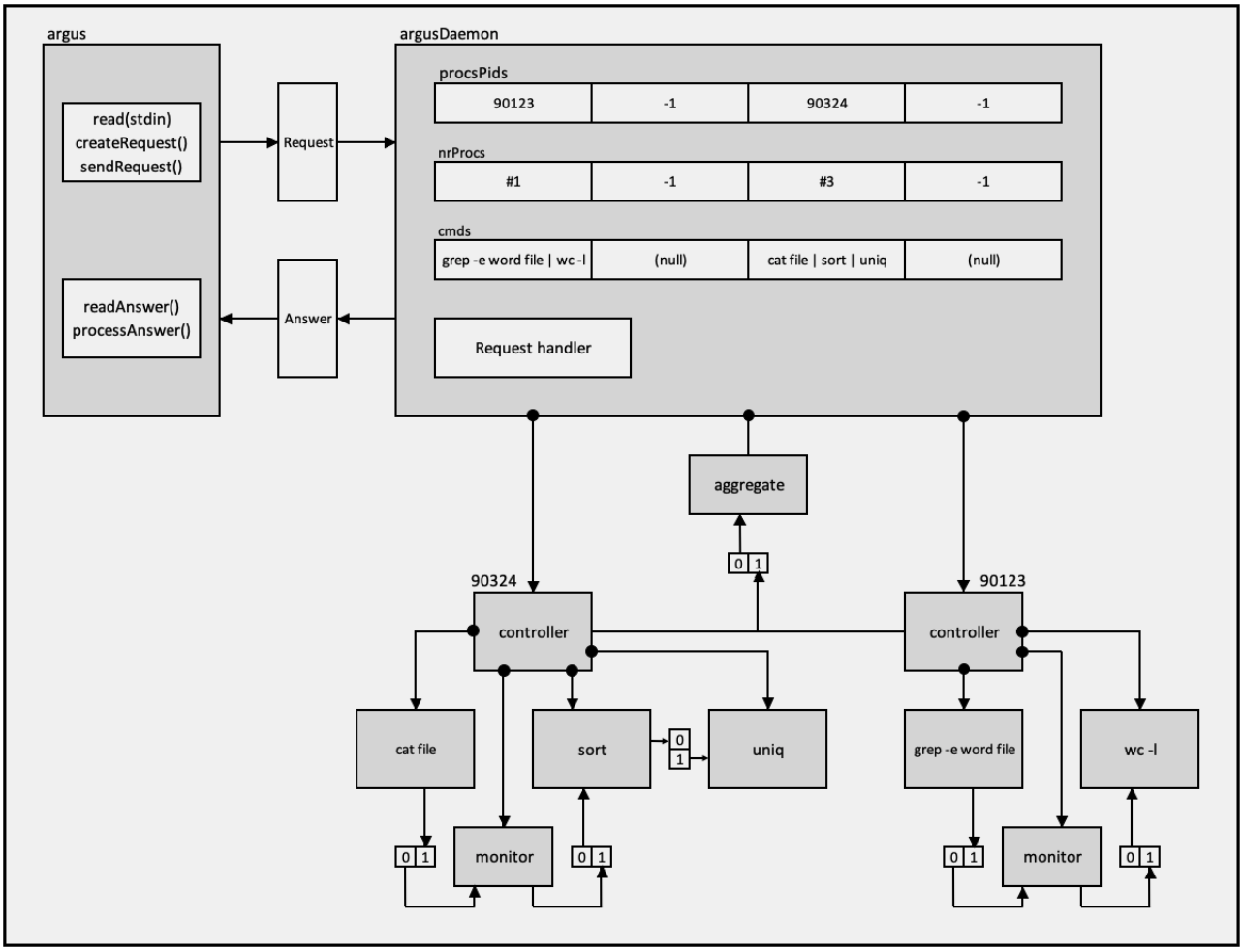

Figura 1: Arquitectura do sistema

4 Concepção/Desenho de resolução

Nesta secção irá ser apresentado, detalhadamente, como o servidor procede para dar resposta a cada um dos comandos definidos. Para além disso, irá ser apresentado também o programa do lado do cliente.

4.1 Cliente (*argus*)

O programa cliente, como já foi dito anteriormente, tem a possibilidade de funcionar modo linha de comandos ou modo *shell*. Esta diferença é feita através no número de argumentos passados ao programa. Se o número de argumentos for maior que um, o programa entra em modo linha de comandos, envia o pedido ao servidor, e pode ou não esperar por uma resposta. Caso o número de argumentos seja igual a 1, o programa entra em modo *shell* e fica à escuta do *stdin* para ler os comandos introduzidos pelo utilizador.

O algoritmo abaixo demonstra, em pseudo-código, o comportamento do programa cliente.

Algorithm 1 client program

```
 1: function ARGUS(argc, argv)
 2:     openFifos(requestFifo, answerFifo)
 3:
 4:     if argc > 1 then
 5:         cmdLn(argv)
 6:     else
 7:         bash()
 8:
 9: cmdLn(argv)
10:     Request r = createRequest(argv)
11:     sendRequest(r, requestFifo)
12:     if needAnswer() then
13:         Answer a = receiveAnswer(answerFifo)
14:         showAnswer(a)
15: bash()
16:     while (cmd = read(stdin)) ≠ "sair") do
17:         Request r = createRequest(cmd)
18:         sendRequest(r, requestFifo)
19:         if needAnswer() then
20:             Answer a = receiveAnswer(answerFifo)
21:             showAnswer(a)
```

Os comandos *tempo-inactividade*, *tempo-execucao* e *terminar* geram mensagens não bloqueantes, uma vez que não esperam por uma resposta do servidor. O comando *execucao* bloqueia à espera de receber o número da tarefa atribuída pelo servidor. O comando *listar* bloqueia à espera de receber a lista das tarefas em execução, sendo que primeiro recebe o número de linhas que vai receber e, de seguida, lê linha a linha, imprimindo para o ecrã o resultado da listagem. O comportamento do histórico é semelhante, recebendo primeiro o número de linhas de histórico que vão ser enviadas e, de seguida, recebe linha a linha, imprimindo para o ecrã. O comando *output*, em vez de receber o número de linhas, recebe o número de *bytes* que vão ser enviados pelo servidor e, de seguida, lê blocos de 512 *bytes* do servidor até receber a mensagem na totalidade. Por fim, o comando *ajuda* não precisa de qualquer interação com o servidor, imprimindo de imediato a *string* de ajuda à execução.

4.2 Servidor (*argusDaemon*)

É o servidor que contém toda a lógica de atendimento de pedidos do cliente. Como se pode observar na *Figura 1*, o servidor é composto por três *arrays* associativos. Um é usado para guardar os *pid's* dos processos criados para executar as tarefas recebidas pelo cliente. Outro é usado para guardar o número da tarefa que um determinado processo está a executar. Por fim, o último *array* é usado para guardar a *string* do comando que está a ser executado. Na *Figura 1*, uma das relações que existe, é, por exemplo:

90123 → #1 → grep -e word file | wc -l

Esta relação permite monitorizar os processos que estão a ser executados no servidor. Quando um processo de execução termina, a informação contida nos *arrays* associativos relativos a essa tarefa é removida e registado o estado de terminação do processo.

Para além destas estruturas, o servidor também mantém em memória o *pid* do processo que está a ser usado para agregar os *output's* de cada tarefa, bem como os tempos de inactividade e de execução.

Para uma melhor percepção do funcionamento do servidor, a *Figura 2* vai servir de auxílio na explicação de certos comandos. As setas normais representam comunicação entre *pipes*, as setas com um ponto no início representam a relação $processoPai \rightarrow processoFilho$ e as setas com um traço no início representam ficheiros abertos pelo processo.

4.2.1 Executar

Quando o cliente pretende executar uma tarefa, é criado um processo, designado por *controller*, responsável por criar a rotina de execução da mesma. O *pid* do processo criado,

o número da tarefa e a *string* da tarefa, são guardados nos *arrays* associativos contidos no servidor.

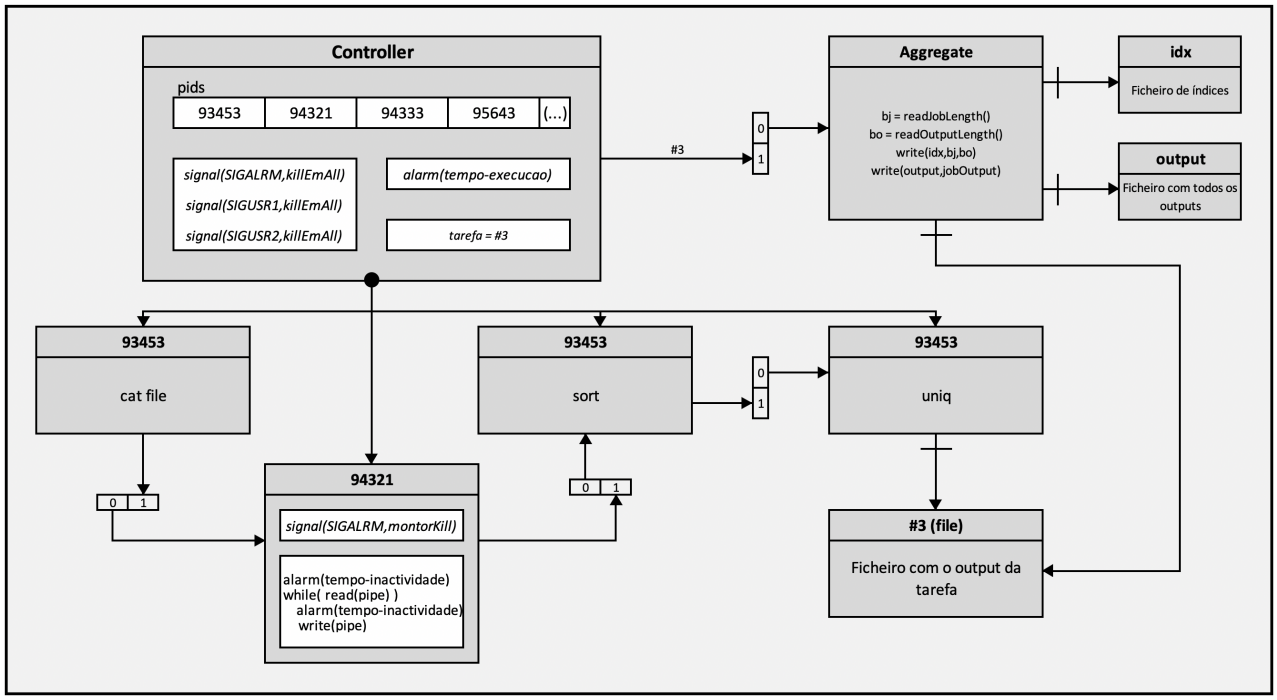

Figura 2: Arquitectura do controlador de execução

Dentro do controlador, existe um *array* que permite manter em memória os *pid's* dos processos criados pelo controlador e, para além disso, existe uma rotina de tratamento para os sinais *SIGALRM*, *SIGUSR1* e *SIGUSR2*. Sempre que estes sinais são lançados, a função de tratamento *killEmAll* é responsável por mandar o sinal *SIGKILL* a todos os processos em execução e, dependentemente do tipo de sinal recebido, define um determinado código de saída do processo que, por sua vez, irá ser usado pelo processo principal do servidor para persistir esses dados no ficheiro de histórico.

Quando é executada a função que dá início à execução da tarefa, é criado um *alarm* com o tempo de execução e, caso este tempo seja ultrapassado, é lançado o sinal *SIGALRM* e tratado. O sinal *SIGUSR1* é lançado quando o cliente pretende terminar uma tarefa em execução e o sinal *SIGUSR2* é usado quando o tempo de inactividade é ultrapassado. Estas duas últimas funcionalidades serão explicadas mais à frente.

O controlador cria sempre um processo a mais, designado de processo de monitorizaçao, para controlar o tempo de inactividade, que irá ser explicado posteriormente.

4.2.2 Tempo de inactividade

Quando o cliente pretende definir o tempo de inactividade, envia o respectivo tempo para o servidor. A partir deste momento, e até um novo valor ser definido, todos os processos criados terão o mesmo tempo de inactividade.

A lógica do tempo de inactividade definida foi, caso o primeiro processo não envie nada para o processo de monitorização no tempo definido, o processo de monitorização lança o sinal *SIGALRM*. A função de tratamento para este sinal envia o sinal *SIGUSR2* ao processo pai, que pode ser obtido usando a função *getppid()*. Após isto, o processo pai (controlador), executa a função *killEmAll*, que já foi explicada na secção anterior.

Num sistema real, seria mais viável criar um processo de monitorização por cada par de processos de execução, no entanto, o implementado foi criar o processo de monitorização apenas no primeiro par de processos.

4.2.3 Tempo de execução

Como acontece com o tempo de inactividade, quando o cliente pretende definir o tempo de execução para o servidor, envia o valor pretendido e, a partir deste momento, todos os processos têm esse valor de tempo de execução. O processo controlador define um *alarm* com o tempo de execução e, caso este tempo seja atengido, é lançado o sinal *SIGALRM* para o controlador e a função *killEmAll* é executada, enviando o sinal *SIGKILL* para todos os processos em execução.

4.2.4 Terminar

Para terminar uma tarefa em execução, o cliente envia o número da tarefa que pretende terminar. O servidor, ao receber o número da tarefa, procura no *array* que contém os números das tarefas em execução a posição que contém a tarefa pretendida. Uma vez encontrada a posição é lançado o sinal *SIGUSR1* ao processo correspondente. O controlador ao receber esse sinal, envia o sinal *SIGKILL* a todos os processos em execução no controlador e termina com um determinado código de retorno.

4.2.5 Listar

Quando o cliente pretende obter a lista de todos as tarefas que estão a executar num determinado momento no servidor, é percorrido o *array* que contém os *pid's* dos processo em execução e, para cada tarefa é criada uma mensagem que será enviada para o cliente contendo o número da tarefa bem como a tarefa que está a executar. A listagem é enviada comando por comando para o cliente sendo que, antes de tudo, é enviado o número de linhas que vão ser enviadas.

4.2.6 Histórico

Para obter um histórico que contém todos os comandos submetidos pelo cliente bem como o estado de terminação da tarefa, foi criada uma função de tratamento para o sinal *SIGCHLD*. Sempre que um processo controlador termina é lançado o sinal *SIGCHLD* para o processo principal. A função de tratamento usa a função *pid_t wait(int *stat_loc)* para saber qual processo terminou e qual o código de retorno. Sempre que um processo de controlo acaba, é removido dos *arrays* associativos toda a informação relativa a essa tarefa. Dependentemente do tipo de retorno do processo, é criada uma mensagem com o estado de terminação da tarefa e essa mensagem é guardada num ficheiro designado por *history*.

Quando o pedido para obter o histórico é enviado pelo cliente, o servidor abre o ficheiro *history* e, envia linha por linha, o conteúdo do ficheiro para o cliente, enviando primeiro o número de linhas que vão ser enviadas.

4.2.7 Output

Para guardar os *output's* de todas as tarefas e, para que esta informação fosse agregada atomicamente num ficheiro apenas, existe um processo dedicado à agregação dos *output's*. Este processo é criado juntamente com um *pipe* anónimo para que os processos que executam as tarefas possam enviar dados para o processo de agregação. Sempre que é criado um processo controlador, este tem acesso à extremidade de escrita do *pipe* do agregador.

Os processos que criam a rotina de execução de tarefas, quando acabam com sucesso, criam um ficheiro temporário, cujo nome é o número da tarefa que está a ser executada, que contém o *output* da tarefa. Quando o processo acaba a escrita do *output* no ficheiro temporário envia o nome do ficheiro que foi criado para o processo agregador de *output's*.

O processo agregador, que está sempre à escuta do *pipe* à espera de receber nomes de ficheiros para agregar, quando recebe o nome de um ficheiro para agregar, calcula o número de *bytes* que contém o ficheiro, o último *byte* do ficheiro *output* naquele determinado momento e adiciona a um ficheiro de índices, designado por *idx*, a associação entre número da tarefa, o *byte* onde começa o seu *output* e o respectivo tamanho. O número da tarefa não é guardada no ficheiro mas é usada para procura pois sabe-se que, para obter a informação no ficheiro de índices da tarefa p, basta fazer $lseek(fd, p*sizeof(int)*2, SEEK\_SET)$ e ler os dois inteiros consecutivamente.

Quando o servidor recebe o pedido para apresentar o *output* de uma determinada tarefa, é identificado através do ficheiro de índices a sua localização no ficheiro de *output's* e a informação é enviada em blocos de 512 *bytes*, no máximo, sendo primeiro enviado o número de *bytes* do *output*.

O algoritmo descrito abaixo demonstra a lógica de partição e envio do *output* de uma determinada tarefa.

Algorithm 2 send output

```
1: function sendOutput(fifo, nrBytes)
2:     write(fifo, nrBytes, sizeof(int))
3:     div = nrBytes/512
4:     res = nrBytes%512
5:     for i = 0 to div do
6:         read(output, buf, 512)
7:         write(fifo, buf, 512)
8:     read(output, buf, res)
9:     write(fifo, buf, res)
```

O cliente, apresenta praticamente a mesma lógica, mantendo um ciclo de leitura para os blocos de 512 *bytes* e uma última leitura para o resto dos dados.

4.3 Comunicação

A comunicação entre o cliente e servidor é feita através de *pipes* com nomes. Para este efeito, e uma vez que estes *pipes* são unidirecionais, foram criados dois para a comunicação. O primeiro, designado por *requestFifo*, é usado para enviar informação do cliente para o servidor e o segundo, designado por *answerFifo*, é usado para enviar informação do servidor para o cliente.

Para ser mais fácil a passagem de mensagens, foram criadas duas estruturas de dados, *Request* e *Answer* pois, uma vez alocadas as estruturas, basta fazer *read*/*write* das estruturas de dados com o respectivo tamanho, isto é:

```
read(requestFifo,request,getRequestSize())
write(answerFifo,answer,getAnswerSize())
```

5 Testes

Para tornar o processo de teste sistemático foi criado, para o programa em modo linha de comandos, uma *script* de *bash* que contém um conjunto de comandos a executar. Estes comandos englobam toda a funcionalidade do sistema. Para o teste do programa em modo *shell* foi criado um programa, designado por *testBash*, que cria um processo que vai executar o

argus e que lê um conjunto de comandos contidos num ficheiro, designado *bashScript*, e envia por um *pipe* anónimo, linha a linha, esses comandos para serem executados. Em ambos os testes, o intervalo de tempo entre cada execução é de 1s.

O *script* abaixo, demonstra os testes feitos no programa a correr em modo *shell*. É de notar que foram realizados testes semelhantes para o programa em modo linha de comandos.

```
executar 'cut -f7 -d: /etc/passwd | uniq | wc -l'
executar 'cat file | sort | uniq'
executar 'cat file | sort | uniq | wc'
tempo-execucao 5
executar './sleep 2 | wc'
executar './sleep 10 | cat file | sort'
tempo-inactividade 1
executar './sleep 10 | cat file'
tempo-execucao 0
tempo-inactividade 0
executar './sleep 3'
executar './sleep 20 | cat file | sort | uniq'
listar
listar
listar
listar
terminar 8
executar 'cat bigFile'
executar './argus -o 9 | wc'
historico
output 1
output 2
output 3
output 10
sair
```

As três primeiras tarefas servem unicamente para verificar a execução de várias tarefas concorrentes. De seguida, testados os tempos de inactividade e execução para verificar se os programas estão a ser interrompidos durante a sua execução. Após isto, foi testado se o programa é capaz de listar tarefas que estão a ser executadas num determinado momento e, para isso, usamos comandos encadeados em que o primeiro é um *sleep*. A partir dái fez-se quatro listagens para verificar se os processos vão acabando. Foi também testada a execução de uma tarefa que processa um ficheiro relativamente grande, 1MB, com o intuito de verificar se esta informação é transmitida corretamente entre os *pipes* do servidor. Um teste interessante que foi feito também foi usar o próprio programa como tarefa, com o objectivo de verificar se o servidor envia o ficheiro de 1MB sem perdas de informção. Para isto foram usados os comando *./argus -o 9 | wc*, em que o primeiro comando envia o *output* do comando 9, correspondente a *cat bigFile*, para o cliente e, de seguida, usa-se o comando *wc* para verificar se o ficheiro foi transmitido na sua totatilidade.

Por fim, imprime-se o resultado do histórico para verificar se os programas terminaram como era suposto e verifica-se o *output* de uns determinados comandos, anteriormente executados. Estes resultados do *output* são então comparados com o resultado da execução do mesmo encadeamento de comandos mas através da *bash* da máquina.

6 Conclusão

O sistema desenvolvido implementa todas as funcionalidades propostas e o código desenvolvido cumpre as regras de encapsulamento e modularidade.

Uma vez que este trabalho envolveu o uso de várias primitivas do sistema *Unix*, foi possível aprofundar o conhecimento sobre as mesmas o que permitiu, também, tornar o processo de *debug* do código mais eficiente. O facto de se perceber como o sistema operativo opera cada uma das chamadas ao sistema e como funcionam as estruturas de dados em *kernel* fez com que alguns erros feitos durante o desenvolvimento da aplicação fossem corrigidos rapidamente, pois era mais fácil perceber a natureza dos mesmos.

Uma das partes que o grupo achou mais interessante fazer, e ao mesmo tempo a que demorou mais tempo a desenvolver, foi o módulo *controller*, que contém a rotina de execução de tarefas. A maior dificuldade neste módulo foi como tornar genérica a execução de comandos encadeados por *pipes*. O erro mais comum que se estava a obter era a não terminação de processos devido a erros no redirecionamento das extremidades de leitura e escrita dos *pipes*. Quando se obteve o redirecionamento correcto, os erros existentes eram devido ao excesso de descritores abertos no mesmo *pipe*. Os processos estavam a bloquear pois haviam vários processos que estavam a apontar para o mesmo *pipe* o que fazia com que os mesmos bloqueassem à espera de ler mais informação.

Por fim, e uma vez feitos os testes de desempenho do sistema, pode-se concluir que o trabalho está funcional e a responder correctamente a cada uma das funcionalidades. Apresenta também uma boa estruturação que torna fácil a sua compreensão e manutenção.