Universidade do Minho

Mestrado Integrado em Engenharia Informática

Sistemas Operativos

Controlo e Monitorização de Processos e Comunicação

GRUPO 101

85308 - Filipe Guimarães
84912 - Joana Afonso Gomes
84167 - Susana Marques

---

14 de junho de 2020

Conteúdo

1 Introdução

Este relatório contém a apresentação do projeto desenvolvido no âmbito da disciplina de Sistemas Operativos ao longo do segundo semestre, do segundo ano, do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

O objetivo do projeto foi implementar um serviço de monitorização de execução e de comunicação entre processos.

Para a sua concretização, desenvolveu-se uma arquitetura com vários processos concorrentes e diversas funcionalidades.

Neste relatório procuramos descrever sucintamente o sistema desenvolvido e as suas principais funções, bem como as decisões tomadas durante a realização do projeto.

2 Descrição do Problema

O projeto proposto requer desenvolver um programa que permita a um utilizador a submissão de sucessivas tarefas, cada uma delas sendo uma sequência de comandos encadeados por *pipes* anónimos. Além de iniciar a execução das tarefas, o programa também deverá conseguir identificar quais as tarefas em execução e as concluídas.

Deverá ainda ser capaz de terminar tarefas que estão a executar, caso não se verifique qualquer comunicação através de *pipes* anónimos ao fim de um determinado tempo, ou caso seja especificado um tempo máximo de execução.

Deverá ser desenvolvido um **cliente** (`argus`) que ofereça uma interface com o utilizador via linha de comandos e um **servidor** (`argusd`) que mantém em memória toda a informação relevante para suportar as funcionalidades pedidas.

Há que implementar a interface com o utilizador contemplando duas possibilidades:

1ª- Através da linha de comandos, indicando opções apropriadas.

2ª- Através de uma interface textual interpretada (`shell`).

Finalmente, tanto o servidor como o cliente deverão comunicar via ***pipes* com nome**.

3 Resolução do Problema

Para a resolução do problema proposto começamos por esquematizar a arquitetura do programa, conforme apresentado na figura seguinte:

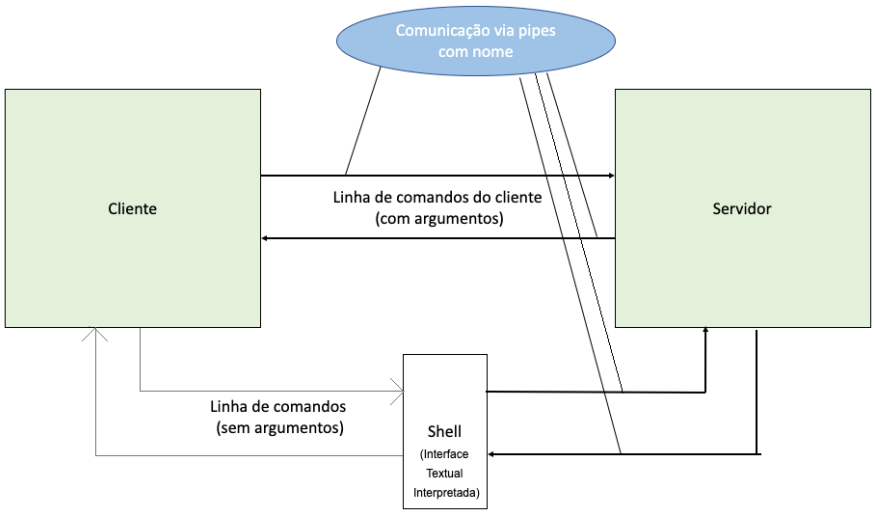

Figura 1: Arquitetura do sistema.

As setas pretas a *bold* na figura representam as ligações (***pipes* com nome**) que foram implementadas para que o cliente (`argus`) e o servidor (`argusd`) possam comunicar entre si.

A `shell` é uma contemplação da interface com o utilizador que existe no cliente, assim como a linha de comandos.

Depois da implentação do sistema, foi ainda utilizado o `teste.sh`, onde fizemos vários testes ao programa de forma simples, rápida e eficiente.

4 Arquitetura

4.1 Argus.c

O módulo `Argus` serve como um cliente de vários "serviços"remotos, oferecendo uma interface com o utilizador via linha de comandos (sem argumentos ou com argumentos-*shell*).

Este módulo apenas comunica com o `Argusd` (servidor) através de ***named pipes***, podendo ser efetuados indicações de funcionalidades de vários clientes que são atendidos pelo `Argusd`, conforme demonstrado na figura seguinte:

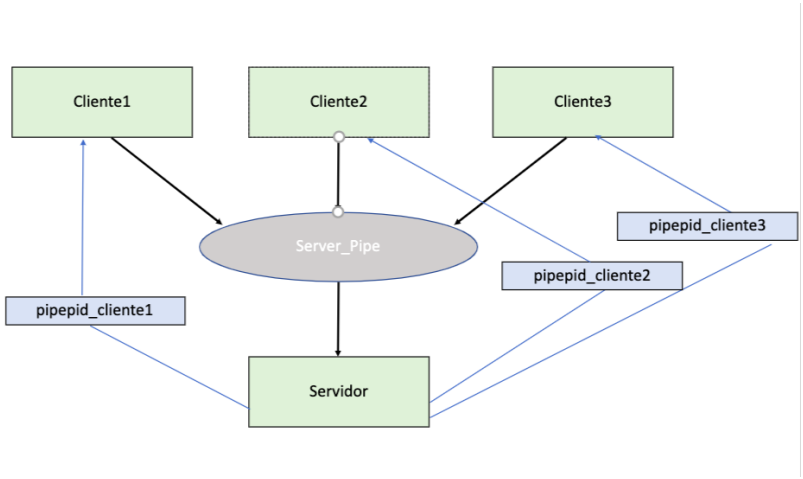

Figura 2: Comunicação entre `Argus` (cliente) e `Argusd` (servidor).

Uma particularidade deste nosso módulo é a capacidade de múltiplos clientes executarem em paralelo, assim foi criado um *pipe* geral: `Server_Pipe`. Este *pipe* recebe os pedidos de funcionalidades de todos os clientes. Contudo, para identificar o cliente a que lhe pertence, primeiro é enviado o seu `pid`. Após implementar a funcionalidade por parte do servidor, este envia uma resposta para um *pipe* individual criado pelo cliente cujo nome é *pipe* + o seu próprio `pid`.

A funcionalidade "apresentar ajuda à sua utilização"(utilização do serviço de monitorização) é a única funcionalidade que não necessita do servidor (`argusd`) para ser implementada, pois apenas pede que sejam apresentados as indicações para aceder às outras funcionalidades. É por isso realizada apenas pelo `argus`, já que a interface pode apresentar estas indicações sem recorrer ao (`argusd`).

4.2 Argus.h

`Argus.h` é o *header* a incluir no cliente (`argus`) e no servidor (`argusd`) e define as estruturas necessárias e utilizadas neste trabalho.

```
1  struct function{
2    //numero da funcionalidade
3    int number;
4    //pid do cliente que envia
5    pid_t client;
6    //pid da tarefa
7    int pid;
8    //qual a funcionalidade
9    int type;
10     int tempo;
11   //numero de comandos
12   int commands_number;
13   //estado da tarefa
14   int state;
15   //lista de comandos a executar
16     struct command commands[COMMAND_NUMBER_MAX];
17   //numero da tarefa
18     int tarefa;
19   //linha para o comando output
20   int line;
21 };
22
23 typedef struct function *FUNCTION;
```

Esta estrutura armazena uma funcionalidade do sistema que pretendemos implementar e utiliza diferentes variáveis dependendo do tipo de funcionalidade a que corresponde (variável `type`). As variáveis que ela contém encontram-se devidamente comentadas, como podemos ver acima, e para além de todas aquelas que têm o seu significado perceptível, também temos uma variável `commands` que representa a lista de comandos a executar. Estes comandos são representados por uma estrutura que está representada a seguir:

```
struct command{
  //estado de cada comando na tarefa
  int state;
  //pid do comando
  int pid;
  //conteudo do comando
  char command[COMMAND_LENGTH_MAX];
};

typedef struct command *COMMAND;
```

A estrutura `command` tem como variáveis o `state`, que corresponde ao estado do comando na tarefa, o `pid`, que corresponde ao *pid* do comando (e que é único), e o `command` que é um `char * (string)` cujo valor máximo é dado por `COMMAND_LENGTH_MAX`;

```
struct outputidx{
  //offset do output
  off_t offset;
  //numero da funcionalidade correspondente
  int function_number;
  //tamanho do output correspondente
  int size ;
};

typedef struct outputidx *IDX;
```

A função da estrutura `outputidx` é de servir como índice no ficheiro de índices. Ao optar por uma estrutura conseguimos guardar o número da funcionalidade a que corresponde este índice bem como o *offset* e o tamanho que esta funcionalidade tem no ficheiro *output*.

4.3 Argusd.c

O módulo `Argusd` é responsável por implementar a API do enunciado, funcionando como o servidor.

Assim, recebe indicação das funcionalidades que o módulo `Argus` lhe passou, implementa-as e emite respostas, comunicando com este via *named pipes*, como vimos anteriormente.

4.4 Implementação das funcionalidades

4.4.1 Definir o tempo máximo de inatividade de comunicação num pipe anónimo

Para o tempo de inatividade definimos um *alarm handler*. Nele percorremos todos os comandos de uma tarefa e verificamos se ainda estão em execução. Se estiverem, enviámos um `SIGTERM` a cada um e matámos também a tarefa atual. O *alarm* é accionado dentro do filho que irá executar um dos comandos da tarefa.

4.4.2 Definir o tempo máximo de execução de uma tarefa

Para o tempo de execução definimos também um *alarm handler*. À semelhança do anterior, percorremos todos os comandos de uma tarefa e verificamos se estão ainda em execução. Se estiverem enviamos um `SIGTERM` a cada um e matamos também a tarefa atual. O *alarm* é accionado dentro do processo onde a tarefa executa.

4.4.3 Executar uma tarefa

Como se pode observar na figura, a execução das tarefas é feita através de *pipes* anónimos. Para a execução de uma tarefa é criado um filho onde ela vai executar e, dentro dele, um outro para cada comando dentro da mesma.

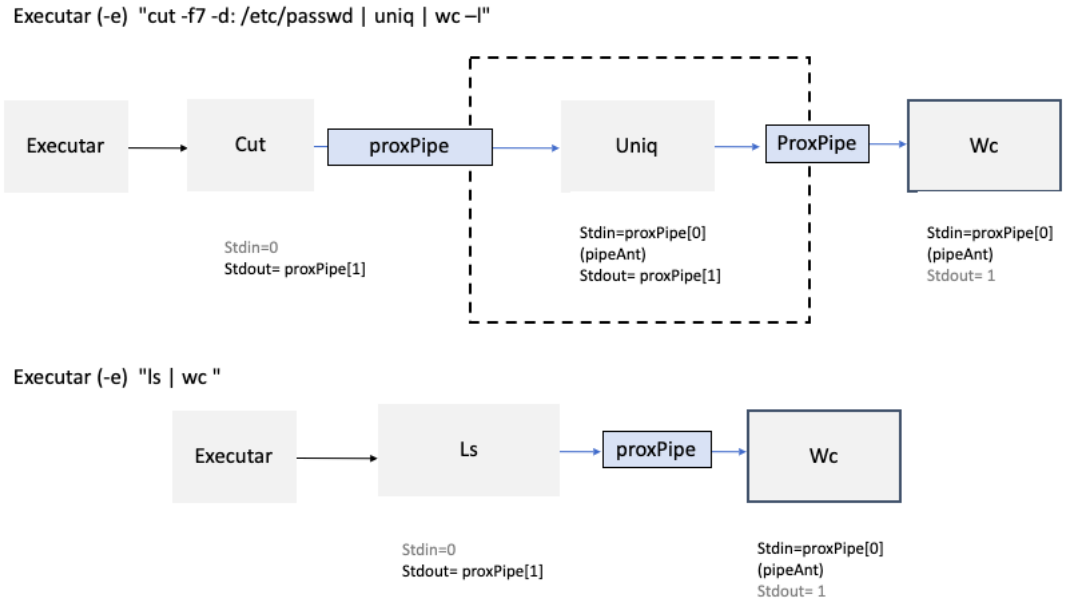

Figura 3: Execução de tarefas com uma sequência de comandos encadeados por *pipes* anónimos

Para o redirecionamento de dados entre os *pipes* temos de observar 3 estados principais que podemos obter: o estado quando entramos no primeiro comando, o estado ao longo da execução da tarefa, e o final quando não temos mais comandos para a frente. No primeiro caso vamos ter com `stdin` o `STDIN_FILENO` e vamos redirecionar o nosso `stdout` para o `proxPipe[1]`, como vemos na imagem, passando a ser o `pipeAnt`. O comando seguinte vai pegar nesse *pipe* e e torná-lo o seu `stdin` passando posteriormente o stdout de maneira equivalente. No último comando o que se altera é que o stdout passará a ser o `STDOUT_FILENO`.

4.4.4 Listar tarefas em execução

Sendo que guardamos um ficheiro com todas as tarefas já executadas e por executar apenas precisamos de percorrer o ficheiro e enviar ao cliente apenas as que têm o estado como [RUNNING].

4.4.5 Terminar uma tarefa em execução

Para terminar uma tarefa temos de a obter a partir do ficheiro onde guardamos todas as tarefas. Para tal fazemos um `lseek` com o número da tarefa multiplicado pelo tamanho da nossa estrutura, poupando comparações desnecessárias. Percorremos então todos os comandos de uma tarefa e enviamos um `SIGTERM` a cada um, matando também a tarefa atual. Alteramos o estado da função e reescrevemos no ficheiro.

4.4.6 Listar registo histórico de tarefas terminadas

A listagem do registo é equivalente à das tarefas em execução imprimindo, desta vez, as tarefas que já não estão a correr.

4.4.7 Consultar *standard output* produzido por uma tarefa já executada (funcionalidade adicional)

Para a consulta do *output* teríamos de encontrar o *index* correspondente à nossa função, para isso percorremos o ficheiro `log.idx` até encontrar o indíce com o mesmo número de tarefa. Após encontrado o índice, temos acesso ao `off_set` e *size* no ficheiro de `output`, lendo então esse pedaço de ficheiro e enviando para o cliente que fez o pedido.

5 Testes

Para assegurar que as secções críticas do programa funcionam corretamente, foram criados **ficheiros de teste**, tais como:

`teste.sh`:

```
#!/bin/bash
./argus -m 5
./argus -e "sleep 3"
./argus -t 1
sleep 1
./argus -e "ls|wc"
./argus -l
./argus -h
./argus -e "date"
./argus -e "cut -f7 -d: /etc/passwd |uniq|wc -l"
./argus -o 2
./argus -e "./time.sh"
sleep 5
./argus -r
exit
```

`teste.txt`:

```
tempo-execucao 5
executar "sleep 3"
terminar 1
listar
executar "ls|wc"
ajuda
executar "date"
executar "cut -f7 -d: /etc/passwd |uniq|wc -l"
output 2
executar "./time.sh"
listar
historico
exit
```

O ficheiro `time.sh` é uma *shell script* criada por nós que apresenta o tempo atual em cada linha do *output* em segundos desde que é executado até que aconteça algo que o faça parar (`ctrl/cmd +c`, funcionalidade terminar, atingir o tempo de execução máximo, *exit*, etc...).

Testamos ainda o programa com múltiplos clientes em simultâneo acedendo ao servidor:

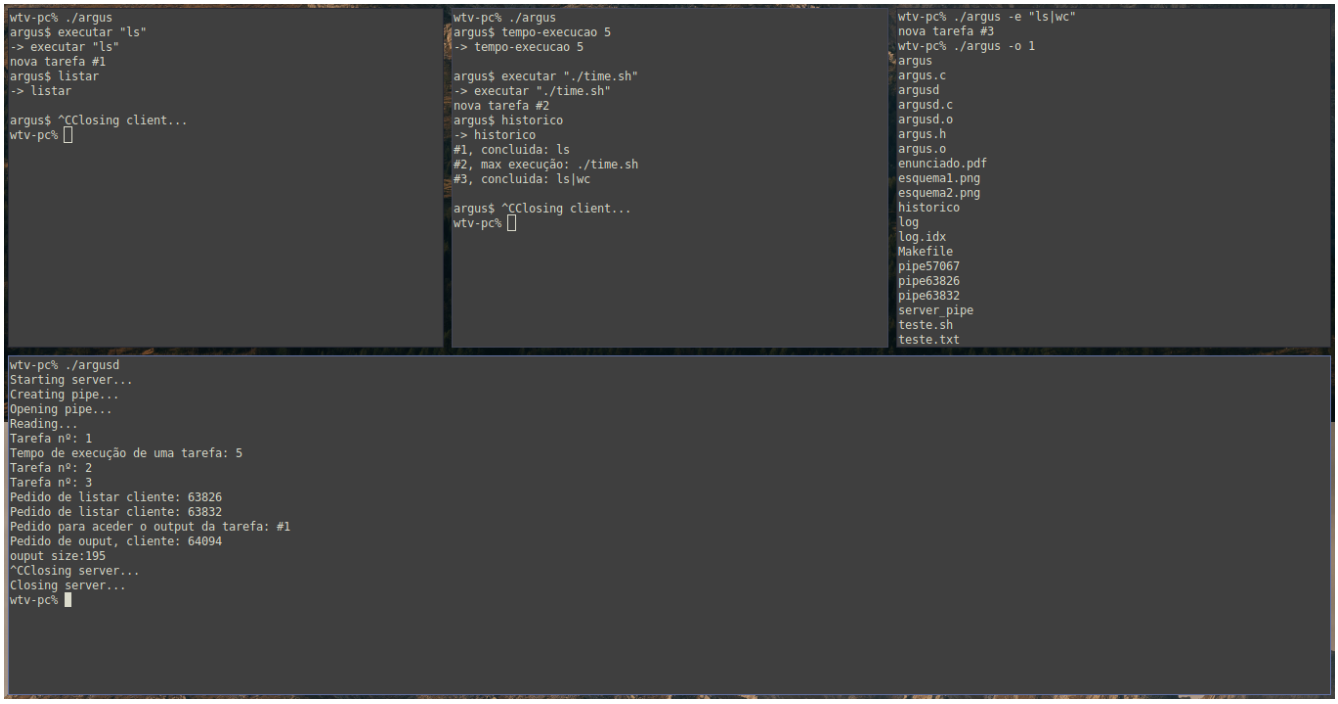

Figura 4: Múltiplos clientes a aceder ao servidor

6 Conclusão

Face ao problema apresentado e analisando criticamente a solução esperada, concluímos que cumprimos as tarefas propostas, conseguindo atingir os objetivos definidos, uma vez que foram implementadas todas as funcionalidades, quer básicas, quer adicionais.

Para conseguir implementar este projeto recorremos a conceitos apresentados nas aulas da unidade curricular, tais como as *system calls* `pipes`, `fork`, `exec`, `read`, `write`, etc., tendo sempre em consideração o melhor desempenho possível dos programas.

Um dos maiores desafios que enfrentamos que foi, porém, ultrapassado, foi a implementação correta e eficiente de tarefas que executam concorrentemente.

Também dirigimos especial atenção ao acesso exclusivo a ficheiros, pelo facto de termos inúmeros problemas se dois programas tentarem escrever ao mesmo tempo para o mesmo ficheiro.

Deste modo o grupo avalia o trabalho de uma forma bastante positiva no que respeita à compreensão dos conceitos abordados e à sua aplicação.