Universidade do Minho
Escola de Engenharia

Controlo e Monitorização de Processos e Comunicação

Sistemas Operativos - Grupo 5

João Pedro da Santa Guedes A89588
Luís Pedro Oliveira de Castro Vieira A89601
Pedro Miguel de Soveral Pacheco Barbosa A89529



14 de junho de 2020

Índice

1 Introdução

Neste semestre, no âmbito da unidade curricular de Sistemas Operativos, foi-nos proposta a realização de um trabalho de modo a pôr em prática toda a aprendizagem sobre sistemas operativos, com auxílio da linguagem de programação imperativa C.

O projeto consistiu na idealização e realização de um programa que implementasse um serviço de monitorização, de execução e de comunicação entre processos. Este deveria permitir ao utilizador a submissão de sucessivas tarefas encadeadas por pipes anónimos. Deverá também, além de inicializar, identificar, averiguar a conclusão, terminar ,caso não se verifique comunicação através dos pipes anónimos ao fim de um dado tempo ou caso o tempo de execução estipulado seja ultrapassado, toda e qualquer tarefa. A interface deste programa deverá disponibilizar duas opções: uma através de linha de comando e a outra através de uma interface textual interpretada, uma shell.

2 Arquitetura do Programa

Após análise do problema proposto, foram idealizados 3 módulos principais e um auxiliar.

Foi criado o módulo **mkfifo**, responsável pela criação dos pipes com nome utilizados na comunicação entre o cliente e o servidor, e vice-versa, e outros processos. Os modulos que servem como a base do nosso programa são: o módulo **argus** onde se encontram as funcionalidades referentes ao cliente e o módulo **argusd** onde se encontram as funcionalidades referentes ao servidor, e onde é tratada a interpretação e execução dos comandos enviados pelo cliente.

O módulo **functions** contém funções gerais que usamos no nosso programa.

2.1 Mkfifo

Este módulo é responsável pela criação dos pipes com nome necessários à execução do nosso programa, nomeadamente o pipe de comunicação entre cliente e servidor , **fifo-cl-sv**, o pipe de comunicação entre o servidor e o cliente, **fifo-sv-cl**, e os pipes que comunicam com o processo pai: o tempo de execução, **pipe-task-executionTime**, o tempo de inatividade, **pipe-task-inactivityTime**, e as tarefas feitas ,**pipe-task-done**.

2.2 Argus

Este módulo representa a interface a utilizar pelo cliente de modo a poder comunicar com o servidor enviando as tarefas que pretende executar, podendo fazê-lo de duas maneiras: ou executa como linha de comando

```
~/Desktop/SO/SO_20/src$ ./argus -e "ls -l | wc "
~/Desktop/SO/SO_20/src$       19     164    1047
```

ou como uma interface textual interpretada.

```
[redacted]:~/Desktop/SO/SO_20/src$ ./argus
exec ls -l | wc -l
19
```

2.3 ArgusD

Por fim, este módulo é o principal responsável pela execução do nosso programa. Recebe a informação vinda do lado do cliente, isto é, recebe as tarefas que ele pretende executar, interpreta-as e executa-as, tudo através da comunicação com auxilío de pipes anónimos e com nome (FIFOs). Quando recebe o comando dado pelo cliente, recorre ao seu interpretador onde avalia a opção desejada e, de acordo com a mesma, remete para a funcionalidade do programa correspondente.

3 Estrutura do Programa

De forma a conseguirmos implementar diversas funcionalidades requeridas pelo enunciado, e após analisarmos o que nos era pedido e à medida que íamos realizando o trabalho, concluímos que iríamos precisar de diversas variáveis **globais**. Além das triviais, tais como descritores de ficheiros e tempos máximos de inatividade e de execução, definimos também as seguintes: ****pidsfilhos** que guarda os processos responsáveis pela realização de uma dada tarefa, **sizeMax** que auxilia à alocação e realocação da nossa estrutura principal, **Tarefa *tarefas** que é um array de estruturas auxiliares ao nosso trabalho, **tar** que dentro do servidor é a próxima tarefa a ser preenchida e que dentro do processo responsável por cada tarefa (e seus filhos) corresponde ao número da sua tarefa e **tarefaRes** que é o processo responsável por uma tarefa (pid do processo).

```c
typedef struct struct_tarefa{
    char *tarefa;
    pid_t pidT;
    int status;
    int o_start;
    int o_size;
}*Tarefa;
```

Idealizámos uma estrutura que nos auxiliasse na realização das mesmas. Assim **Tarefa** contém: um **char* tarefa** referente ao comando introduzido pelo cliente, um **pid-t pidT** referente ao processo responsável pela execução de uma dada tarefa, um **int status** referente ao estado da tarefa, um **int o-start** e **int o-size** referentes à posição no ficheiros logs.txt e o tamanho do seu output, respetivamente.

NOTA: status será 0 quando a tarefa não é válida, 1 quando esta está em execução, 2 quando foi completada, 3 e 4 quando excedeu o tempo máximo de inatividade e de execução, respetivamente, e 5 quando foi terminada pelo cliente.

4 Funcionalidades do Programa

4.1 Definir o Tempo Máximo de Inatividade

4.1.1 Implementação

O programa tem como pré-definição um tempo de inatividade infinito, isto é, as tarefas podem continuar a executar, mesmo que não haja atividade nos pipes que entre si estão a comunicar. No entanto, este pode ser alterado. Como se trata de uma variável global, torna-se relativamente fácil de a alterar. Assim apenas implementamos uma função que recebe o input do cliente com o tempo máximo de inatividade a definir, e atribuímos esse valor à variável global.

4.1.2 Exemplo de execução

$./argus - i$ **(tempo)**

Aqui, (*tempo*) deverá ser substituído pelo valor pretendido para o tempo de inatividade, em segundos.

4.2 Definidir o Tempo Máximo de Execução

4.2.1 Implementação

O programa tem como pré-definição um tempo de execução infinito, isto é, as tarefas podem continuar a executar até que terminem. No entanto, este pode ser alterado. Como se trata de uma variável global, torna-se relativamente fácil de a alterar. Assim apenas implementamos uma função que recebe o input do cliente com o tempo máximo de execução a definir, e atribuímos esse valor à variável global.

4.2.2 Exemplo de execução

$./argus - t$ **(tempo)**

Aqui, (*tempo*) deverá ser substituído pelo valor pretendido para o tempo de execução, em segundos.

4.3 Executar uma Tarefa

4.3.1 Implementação

Após receber o input do utilizador referente à opção de executar uma tarefa , $(-e)$, obtemos o que se encontra à frente, isto é, os comandos que o cliente pretende realizar. Começamos por copiar o descritor de ficheiro do pipe que comunica entre o servidor e o cliente para o standart output. De seguida criamos um processo filho, o qual está responsável pela execução da tarefa pedida. Para tal recorre à função $exec - pipe$, à qual passamos uma string com as tarefas pedidas pelo utilizador.

Na função $exec-pipe$ começamos por percorrer a string com os comandos pedidos pelo cliente, contabilizando quantos são, para a seguir separá-los de forma a serem executados. Esta função cria filhos concorrentemente de forma a que cada um execute um dos comandos, comunicando entre si através de pipes anónimos, de forma a obter o output expectado.

Enquanto isso, no processo pai, são atualizadas as informações sobre essa tarefa, nomeadamente o processo que a está a realizar, o estado em que se encontra e qual a tarefa executada.

Para guardar esta informação utilizamos uma estrutura auxiliar denominada **Tarefa**.

4.3.2 Exemplo de execução

$./argus - e$ **"(comandos)**

De notar que quando pretendemos comunicar com o servidor utilizando linha de comando, devemos colocar aspas entre o primeiro e último comando! Quando estamos a utilizar a interface textual interpretada as aspas já não são necessárias.

Aqui, (*comandos*) deverá ser substituído pelos comandos que pretende realizar.

4.4 Listar Tarefas em Execução

4.4.1 Implementação

Quando o cliente pede ao servidor a lista de tarefas em execução, é chamada uma função auxiliar $printaTarefasEmExecucao$ que percorre o array de estruturas Tarefa e transmite ao cliente, através do pipe que comunica entre o servidor e este, as tarefas que se encontram em execução e o seu número único.

4.4.2 Exemplo de execução

$./argus - l$

4.5 Terminar uma Tarefa em Execução

4.5.1 Implementação

Para terminar uma tarefa, o cliente deve indicar o número único da tarefa que pretende executar. Quando é pedido o término de uma tarefa é chamada a função $terminaTarefa$ que recebe o número único da tarefa que se pretende terminar. Esta, com o uso do número único da tarefa, obtém o pid do processo que se encontra a realizar essa tarefa, recorrendo ao array de Tarefa, e envia um sinal SIGUSR1 que é tratado pelo $killProcessUSR1 - handler$. Este, por sua vez envia um sinal SIGKILL a todos os seus processos filhos e de seguida termina a sua execução.

4.5.2 Exemplo de execução

$./argus - t$ **(num Tarefa)**

($numTarefa$) deverá ser substituído pelo número único da Tarefa que se pretende terminar.

4.6 Listar Histórico de Tarefas Terminadas

4.6.1 Implementação

Quando o cliente pede ao servidor o histórico de tarefas terminadas, é chamada uma função auxiliar *printaHistorico* que percorre o array de estruturas Tarefa e procura aquelas que não estão em execução, ou seja, que já foram concluidas, ou que excederam ou o tempo máximo de execução ou o tempo máximo de inatividade. De seguida transmite ao cliente, através do pipe que comunica entre o servidor e este, o histórico das tarefas dadas por terminadas.

4.6.2 Exemplo de execução

$./argus - r$

4.7 Apresentar Ajuda à Utilização

4.7.1 Implementação

Quando o cliente pede ao servidor ajuda na execução do programa, ou seja, pretende saber os comandos disponíveis, é utilizada a função *printaAjuda* que imprime um menu com os comandos disponíveis e os argumentos necessários, quer para a linha de comando, quer para a interface textual interpretada.

4.7.2 Exemplo de execução

$./argus - h$

4.8 Funcionalidade Adicional

4.8.1 Implementação

Preenchimento do ficheiro logs.txt: Para a realização desta tarefa adicional, aquando da execução das tarefas pedidas pelo utilizador o seu output será transmitido tanto ao cliente como a um ficheiro temporário com a ajuda da função *tee* do Linux. Posteriormente a tarefa ao ser dada como terminada, vai enviar um sinal SIGUSR1 ao pai, que neste caso é o servidor, para proceder à comunicação com ele. O servidor vai receber a tarefa em questão, abrir o ficheiro temporário, abrir o ficheiro logs e escrever no ficheiro de logs o conteúdo que está no ficheiro temporário. Enquanto isso, vai atualizando variáveis locais para que no final atualize a estrutura Tarefa, com a posição inicial no ficheiro de logs e o tamanho que ocupa.

Leitura do ficheiro de logs.txt: Nesta fase é chamada a função *printaOutput* que usando a estrutura Tarefa, obtém a posição inicial e o tamanho do output para proceder à escrita do mesmo.

4.8.2 Exemplo de execução

$./argus - o$ **(num Tarefa)**

Note que em $(numTarefa)$ deverá ser colocado o número da tarefa que pretende obter o output.

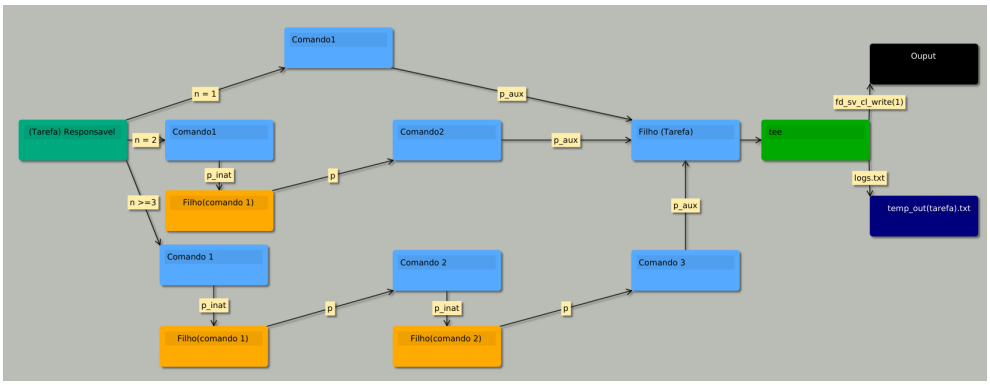
5 Execução e Inatividade

Conforme estipulado no enunciado, a comunicação e execução de tarefas através dos pipes, poderá ser limitada por dois fatores: tempo máximo de execução e tempo máximo de inatividade. Assim é preciso ter em mente estes fatores aquando do pedido pelo cliente na execução das tarefas, ou seja, caso exceda um dos tempos estipulados, a tarefa deve terminar e ser sinalizada de acordo com a forma como foi abruptamente terminada.

Execução

Quando executamos uma tarefa, a nossa função $exec - pipe$ cria filhos de forma concorrente, e após a criação dos mesmos envia um alarme que caso seja recebido pelo processo, é tratado com a função $sigExecutionAlarmHandler$. Esta envia um sinal ao pai a informar que vai comunicar, abrindo uma das extremidades do **FIFO** $pipe - task - executionTime$, enviando a tarefa em questão ao pai para que este seja atualizada na nossa estrutura e enviando a seguir um sinal **SIGKILL** aos processos responsáveis pela execução dos comandos, terminando assim a tarefa.

Inatividade



Quando uma tarefa é executada e o tempo de inatividade foi previamente definido para um valor maior que zero, cada processo filho responsável pela execução de um comando, cria um novo processo filho responsável tanto pelo alarme como pela transmissão de informação para o processo filho que executa o comando seguinte. Ou seja, cada vez que passa informação no pipe é dado um novo alarme. Quando este é sinalizado é tratado pela função $warnParentInactivityHandler$ que, por sua vez, envia um sinal **SIGINT** ao processo responsável pela execução da tarefa para avisar que vai ser terminada pois ultrapassou o tempo máximo de inatividade. Por sua vez este processo sinaliza ao pai que vai comunicar, abrindo uma das extremidades do **FIFO** $pipe - task - inactivityTime$, enviando a tarefa em questão ao pai para que esta seja atualizada na nossa estrutura e enviando a seguir um sinal **SIGKILL** aos processos responsáveis pela execução dos comandos, terminando assim a tarefa.

6 Conclusão

Ao longo deste projeto, apesar de sermos capazes de implementar muitos dos nossos conhecimentos sobre Sistemas Operativos, que foram adquiridos nas diversas aulas, sentimos , mesmo assim, diversas dificuldades que nos obrigaram a ser um pouco mais autónomos e a procurar diversas soluções para os problemas que nos foram apresentados. Uma das grandes dificuldades que sentimos foi conseguir implementar uma comunicação funcional entre o cliente e o servidor de forma a não deixar processos órfãos. Em relação às funcionalidades que nos foram propostas implementar, aquela em que sentimos maior dificuldade foi referente ao tempo por inatividade, isto é, sermos capazes de terminar uma tarefa por exceder este tempo. Ainda assim achamos que realizámos um bom trabalho, sendo capazes de implementar todas as funcionalidades previstas pelo enunciado, inclusivé a adicional, e, por termos de nos tornar mais autónomos de forma a suceder no trabalho, enriquecemos o nosso conhecimento sobre Sistemas Operativos.