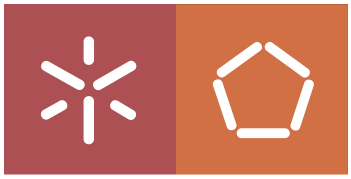

Universidade do Minho
Escola de Engenharia

Sistemas Operativos

Controlo e Monitorização de Processos e Comunicação

14 de Junho de 2020

Grupo 19

José Magalhães,A85852
Ivo Baixo,A86579
Jorge Vieira,A84240

Conteúdo

1 Introdução

No âmbito da unidade curricular de Sistemas Operativos, foi proposto desenvolver um projeto que permita a implementação de uma plataforma para Controlo e Monitorização de Processos e Comunicação sob a forma de cliente/servidor utilizando, para tal efeito, *pipes* com nome *FIFO*.

Ao longo deste relatório será possível ver, clarificadas, as escolhas e decisões tomadas pelo grupo de modo a obter a melhor solução possível para o que nos foi proposto.

2 Servidor

O servidor é a parte principal do nosso trabalho, sendo o elo de ligação entre todos os módulos do mesmo. É imperativo que este esteja em execução antes de qualquer interação do cliente pois, caso contrário, não é assegurada a execução das tarefas pedidas.

Além do que foi dito é, também, responsável pela criação dos ficheiros e atualização dos mesmos.

2.1 Modo de funcionamento

De um modo geral, ao servidor cabe-lhe ler e interpretar os comandos vindos do cliente, recebidos através da leitura de um *named pipe(**FIFO**)*, cujo conteúdo fora escrito anteriormente pelo já dito cliente.

De realçar que a mensagem recebida contém, também, o **PID** do cliente, para que, mais tarde, o servidor possa enviar informações ao cliente, de modo exclusivo. Podemos, então, ver o processo da seguinte forma:

1. Ao ser iniciado, mantém-se em *loop*, de modo a ler n comandos;
    (a) Os comandos podem ser obtidos por dois métodos de *input* diferentes, estando assim de acordo com o enunciado, sendo atribuída a estes uma *flag* - 0 ou 1 - que será útil mais à frente;
2. Inicializa a classe *SmartArray*, cujo propósito veremos mais à frente;
    (a) Uma estrutura criada a pensar na funcionalidade mínima 'Listar tarefas em execução' que, ao longo do processo, vai ser executada, como veremos detalhadamente mais à frente;
3. Abre o *named pipe "fifoF1"* apenas com a flag **O_RDONLY**;
    (a) É também responsável pela sua criação;
4. Procede à leitura do ponto referido anteriormente para um *buffer*;

5. Interpreta a *string* recebida, agindo, de seguida, consoante o pretendido pelo cliente:
   (a) É efetuado o *parsing* da linha recebida, resultando este este em quatro campos:
      i. **PID** do cliente;
      ii. Comprimento da *string* introduzida pelo cliente;
      iii. Método de *input*
      iv. *String* introduzida.
   (b) Caso a *flag* - presente no ponto **iv** - introduzida seja **executar** ou **-e** então esta é novamente submetida a um novo *parsing*, de forma a guardar os vários comandos por *pipes*, para posteriormente serem tratados no módulo 'executa', explicado mais à frente.
6. Abre o *named pipe* "**PID**# *pidcli*"apenas com a flag **O_WRONLY** e procede ao envio da resposta ao cliente;
   (a) Este canal de comunicação permite ao cliente ter uma resposta exclusiva, tendo assim 'privacidade' em relação a outros clientes;
   (b) Por decisão do grupo, por cada comando que o cliente mande irá sempre obter uma resposta por parte do servidor. Resposta essa que pode variar, tendo em conta o seguinte:
      i. Nas operações em que são pedidas ao servidor uma resposta por parte do mesmo, como, por exemplo, a opção de 'Listar tarefas em execução', é enviada a resposta pedida;
      ii. Nas restantes operações é enviada uma mensagem de sucesso, como, por exemplo, "Nova tarefa #1", correspondente ao caso em que o cliente executa a primeira tarefa.
7. Liberta a memória alocada durante este processo;
8. Ao longo de todo este processo sequencial, o servidor também pode proceder ao tratamento de sinais, entre eles:
   (a) **SIGUSR2**
      - É invocado no fim do ponto **5** referido no ponto acima. Aquando a execução da tarefa pretendida, é escrito num *pipe* anónimo o número da tarefa correspondente para posteriormente ser lido pelo *handler*, que quando invocado, remove essa tarefa da lista de tarefas em execução;
   (b) **SIGINT**
      - É invocado quando o utilizador pretende terminar uma determinada tarefa, e tem o intuito de enviar este sinal ao processo correspondente à tarefa dada, a fim de o terminar. Utilizou-se **SIGINT** pois era o sinal que tinhamos disponível.

3 Cliente

Neste módulo, em primeiro lugar é criado um **FIFO** denominado com o padrão "**PID**# *pidcli*". De seguida, procede-se à abertura do já anteriormente referido **FIFO** *"fifof1"*, mas apenas em modo de escrita.

Posteriormente o processo é simples. Para o método de *input* em que *argc* é 1, é criado um **processo filho** para cada funcionalidade que o cliente pretenda usufruir, acabando esta por ser lida para um *buffer* e enviada para o servidor para ser executada.

De realçar que a mensagem que é enviada é de seguida construída e corresponde aos campos ditos no subtópico **5 a)** do tópico anterior. O **processo pai**, por sua vez, fica à espera da resposta proveniente do servido. Quando a obtém, lê a mesma, utilizando o canal de comunicação exclusivo, o *named pipe* **PID**# *pidcli*"e, por último, "imprime no ecrã"o resultado.

Para o método em que os argumentos constam todos na mesma linha de comandos, o procedimento é semelhante ao descrito anteriormente, com a diferença de que em vez de ler a linha para um *buffer*, fazemos usufruto do *array argv[]* e, sendo este devidamente tratado e enviado.

4 Comunicação entre cliente/servidor

De seguida podemos ver um esquema de como é efetuada a comunicação no nosso programa, já explicada de forma detalhada anteriormente.

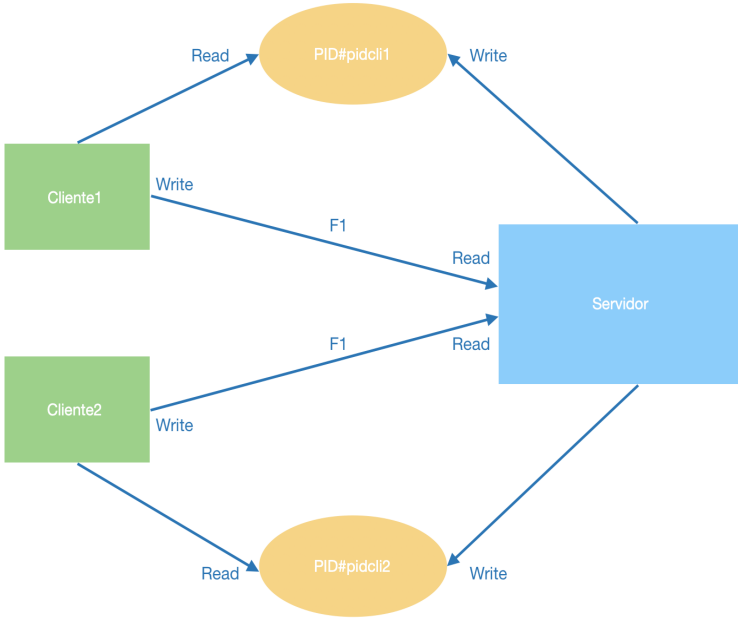

Figura 1: Esquema de comunicação cliente/servidor

5 Funcionalidades mínimas

5.1 Tempo máximo de inatividade

Antes demais é importante referir que considera-se como tempo de inatividade de um pipe anónimo todo o tempo que se passou desde que este foi criado até que se escreveu alguma coisa neste.

A solução adotada para conseguir limitar o tempo de inatividade nos *pipes* anónimos foi, para cada *pipe* anónimo, criar um processo *monitor* que conta o tempo durante o qual o *pipe* não tem lá nada escrito. Para efetuar esta contagem de tempo lança-se um *alarm* e depois cancela-se assim que seja lido pelo menos um *byte* do *pipe* que está a ser monitorizado.

Caso o tempo definido seja excedido, então todos os processos filhos relativos à tarefa em questão serão "mortos", retornando assim o motivo pelo qual a tarefa foi terminada. Para explicar melhor a nossa abordagem, criou-se o seguinte esquema:

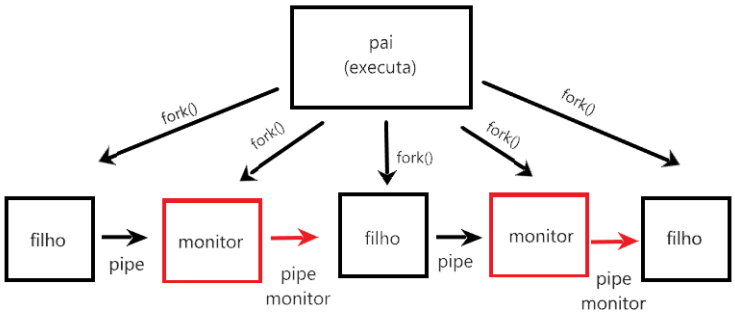

Figura 2: Método utilizado para resolução

5.2 Tempo máximo de execução

Para limitar o tempo de execução de uma tarefa, imediatamente antes de executar uma nova tarefa lança-se um *alarm(n)*, onde n representa o tempo máximo de execução desejado. Assim, se o processo que executa a tarefa recebe um sinal de alarme é porque excedeu o limite de tempo de execução, sendo assim devidamente terminado.

5.3 Execução de uma tarefa

Em primeiro lugar convém realçar que, de modo a garantir que várias tarefas possam executar ao mesmo tempo, para cada tarefa introduzida pelo cliente é criado, no servidor, um **processo filho** para efetuar a execução da mesma garantido, assim , o propósito deste trabalho.

Para executar uma tarefa é necessário primeiro separar a tarefa em comandos e posteriormente executar esses comandos sequencialmente, criando um filho para executar cada um dos mesmos, e garantindo que o pai espera pelo filho anterior antes de criar o próximo. Toda esta execução sequencial é feita recorrendo a *pipes* anónimos.

Criou-se assim a função *executa* com o comportamento descrito em cima, capaz de executar qualquer tarefa fornecida, desde que com comandos e argumentos válidos.

5.4 Listar tarefas em execução

Para uma resolução mais eficaz foi necessário criar uma nova estrutura de dados, um *SmartArrayS* cuja definição é a seguinte:

```
    struct SmartArrayS{
    int tam;
    char **array;
};
```

Optou-se por criar a *struct* acima referida pois o processo que permite adicionar e remover *strings* de um *array* é bastante simples, sempre que uma tarefa inicia e acaba, respetivamente. Desta forma conseguirmos ter acesso às tarefas que estão em execução no momento.

Quanto à sua implementação, quando a tarefa termina é escrito o número da mesma num *pipe* anónimo, sendo que de seguida é lançado um **SIGUSR2** que posteriormente será tratado por um *handler*. Este, por sua vez, vai ler do já falado *pipe* o número da tarefa que terminou e procede à sua remoção no *array* dinâmico

Por fim, quando o utilizador pretende listar as tarefas em execução, então basta ao servidor aceder ao *SmartArrayS* e enviar o seu contéudo para o cliente, sendo este depois impresso no ecrã.

5.5 Terminar tarefa

Para terminar uma dada tarefa começou-se por guardar no servidor, mais propriamente num *array* de *pids*, para cada tarefa, o *pid* do processo que a está a executar. Assim, quando o utilizador deseja terminar uma determinada tarefa n, apenas é preciso enviar um sinal ao processo que está a executar a tarefa em questão e, por sua vez, este terminará.

De realçar que este sinal é, depois, tratado no módulo *executa* recorrendo para isso a funções já criadas para a resolução dos pontos **5.1** e **5.2** tornando, assim, mais simples, a sua implementação.

Acrescenta-se que esta medida permite a que, quando uma tarefa é terminada, todos os processos que resultaram desta também o são.

5.6 Histórico de tarefas

Para a resolução do tópico em questão, o procedimento utilizado foi relativamente simples. Visto todas as tarefas, quando terminada a sua execução, são gravadas num ficheiro denominado "tarefas.txt", então, apenas foi necessário abrir um descritor de leitura para esse mesmo ficheiro e, utilizando a função **readln2(...)**, ler as várias linhas, concatenar as mesmas e enviar a construção resultante para o cliente.

Salienta-se que, por cada linha lida é adicionado um delimitador, de forma a ser mais simples, na parte do cliente, apresentar corretamente todo o histórico presente no ficheiro.

5.7 Ajuda à utilização

Esta funcionalidade foi realizada com alguma facilidade, tendo sido apenas necessário construir uma *string* com o conjunto de funcionalidades do nosso programa e de seguida enviada para o cliente, através dos meios de comunicação já falados.

Realça-se que o modo de construção da *string* varia com o tipo de *input*.

6 Funcionalidade adicional

O grupo, devido a várias razões mas sobretudo a falta de tempo, não foi capaz de concluir a tarefa extra pedida, embora a tenha idealizado, ideia essa que corresponderia aos seguintes tópicos:

- Antes da escrita do *output* da tarefa no ficheiro *output.txt*, calcularíamos o *offset* atual do ficheiro;
- De seguida, utilizando o *whence SEEK_END* da última tarefa antes da tarefa em questão saberíamos onde escrever a próxima tarefa;
- Então, esses *bytes* escritos seriam escritos num ficheiro *"log.idx"*;
- Assim, após estes processos, o ficheiro ficaria já atualizado;
- Posteriormente, quando o cliente invocasse o comando de consultar o ficheiro e para uma procura mais rápida no *".idx"*, utilizaríamos o "truque"de fixar o número de bytes;
- De seguida, tendo os valores do *offset*, e utilizando *lseek(...)* o grupo achou que conseguiria facilmente chegar ao pretendido.

Acrescenta-se em forma de nota que, para corrigir o problema dos *outputs* poderem ser escritos desordenadamente o grupo, como forma de solução, idealizou, no ficheiro *".idx"* ter não só os valores do *offset*, mas também o número da tarefa correspondente.

7 Discussão dos resultados

Nesta secção iremos abordar, de uma forma geral, os resultados obtidos e algumas escolhas do grupo , bem como os testes que usámos para verificar se as funcionalidades funcionavam corretamente.

Uma das principais escolhas do grupo foi a criação do módulo *executa*. Neste módulo consta todo o processo não só da execução de uma tarefa, mas também todo o código necessário para concluir as funcionalidades respetivas aos tempos de execução e tempos de inatividade. Esta decisão baseou-se sobretudo de modo a facilitar um eventual processo de *debug* pois, no entender do grupo, dividir o código por diferentes módulos resulta numa procura mais rápida de erros.

Uma outra escolha que é necessário referir é o modo de comunicação cliente/servidor. A criação de dois canais, um deles público e o outro privado, foi pensada para que a estrutura suportasse vários clientes. Quando nos apercebemos que era apenas necessário suportar um cliente, decidimos manter o já anteriormente feito.

Quanto à parte dos testes utilizámos vários *scripts*, tal como sugerido pela equipa docente. Estes *scripts*, que constam no ZIP do trabalho prático, testam detalhadamente:

1. ***Script 1*** Testa tempo máximo de execução;
2. ***Script 2*** Testa em vários casos os tempos de execução e inatividade;
3. ***Script 3*** Testa listar tarefas em execução e terminar uma delas;
4. ***Script 4*** Testa tempos de inatividade entre *pipes*.

8 Conclusão

Quanto à parte pedagógica, concluímos que este projeto foi muito enriquecedor para o nosso coletivo, pois este trabalho permitiu que melhorássemos as nossas competências a nível prático relacionadas com a UC de Sistemas Operativos.

Reconhecemos que o projeto foi feito com alguma dificuldade pois foi necessário aplicar conteúdo abordado nas aulas práticas que não estava completamente cimentado e clarificado em cada um de nós.

O grupo assume que a parte mais desafiante deste trabalho foi o tópico da inatividade entre os *pipes*, e que foi com alguma dificuldade que chegámos à solução final, que consideramos bastante satisfatória. Os restantes tópicos já iam mais ao encontro do que foi feito nos guiões ao longo do semestre e, por isso, de dificuldade menos elevada.

Em suma, o esforço coletivo foi grande com o intuito de garantir boas soluções para o enunciado proposto, indo assim ao encontro dos objetivos definidos pelo grupo.