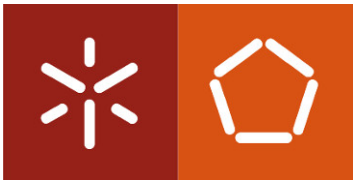

Controlo e Monitorização de Processos e Comunicação

Sistemas Operativos

João Manuel Silva de Amorim - A89466
Rodrigo Caldas Meira - A89511
Rúben Daniel Almeida Adão - A89545

Junho de 2020

Mestrado Integrado em Engenharia Informática
Universidade do Minho

Conteúdo

1 Introdução

Como projeto para a cadeira de Sistemas Operativos, foi nos dado o objetivo de construir um serviço de Controlo e Monitorização de Processos e Comunicação em C. Este serviço deverá ser capaz de executar tarefas sucessivas, conseguindo identificar as tarefas que o mesmo está a executar e terminá-las. Terá também duas interfaces: uma que funciona através de linha de comando, a outra que funciona como uma shell e recebe instruções através do standard input. Iremos ao longo deste relatório, descrever as técnicas utilizadas pelo grupo, para responder a estes problemas.

2 Objetivos

Para este projeto, os objetivos do grupo foram criar um serviço que implementasse as funcionalidades descritas no enuciado (mínimas e adicionais), tendo sempre em vista que este teria duas interfaces:

- Definir um tempo máximo de inatividade entre *pipes* .
- Definir um tempo máximo de execução de um programa.
- Capacidade de executar qualquer comando.
- Listar todas as tarefas em execução.
- Terminar uma tarefa em execução.
- Listar histórico de tarefas terminadas.
- Apresentar ajuda à sua utilização.
- Consultar o *standard output* produzido por uma tarefa já executada.

Era também expectável uma arquitetura de servidor-cliente gerida através de *pipes* com nome.

3 Arquitetura final do Projeto

3.1 Servidor

O Servidor é responsável tanto por executar os comandos introduzidos pelo utilizador através do Cliente, como de criar os *pipes* com nome que serão utilizados para comunicar com essa tal interface Cliente. Dois *pipes* são criados, um para receber informação (comandos do utilizador) e outro para enviar informação ao Cliente, após a execução de um comando.

3.2 Cliente

O Cliente é criado com base na sua interação com o Servidor. O papel do Cliente nesta interação é receber intruções ao vivo (shell) ou comandos para executar no Servidor, escrevendo aquilo que o utilizador lhe fornece, no *pipe* com direção ao Servidor. Simultaneamente, cada vez que obtém informação do Servidor imprime-a no ecrã.

3.3 Módulos

- **server.c** - Codifica as funcionalidades do Servidor.
- **client.c** - Codifica as funcionalidades do Cliente.
- **run.c** - Código do executável responsável por executar um comando.
- **checkpipe.c** - Código do executável responsável por verificar a atividade entre dois processos.
- **collection.c** - Código que implementa as bases de dados utilizadas pela aplicação (listas ligadas e structs auxiliares).

3.4 Ficheiros Adicionais

Ficheiros gerados ao longo da execução do programa.

- **log** - *Output* das tarefas terminadas.
- **log.idx** - Localização do *output* das tarefas, no ficheiro log, por id.
- **history** - Histórico das tarefas terminadas.
- **Ficheiros especiais** - *Pipes* com nome.

4 Problemas e Soluções

Passamos agora a descrever as soluções utilizadas para resolver cada um dos objetivos propostos pelo enunciado. Algumas considerações adicionais:

O Servidor e o Cliente são executáveis separados, tal que o Servidor não necessita de um Cliente aberto para ser executado. Assim, o Servidor continua a correr independentemente da execução ou término de um Cliente, e apenas termina com a ordem explícita do "administrador". O Cliente, por outro lado, necessita de um Servidor aberto para o qual pode enviar comandos.

É de se notar também que cada tarefa é equivalente ao um processo, que iremos denominar de *processo-tarefa*, tal que este possui processos filhos que representam os comandos de uma tarefa, i.e a tarefa "ls -l | cat | wc -c"possui três[1] processos filhos, um responsável por executar o comando "ls -l", outro o "cat", e o último o "wc -c".

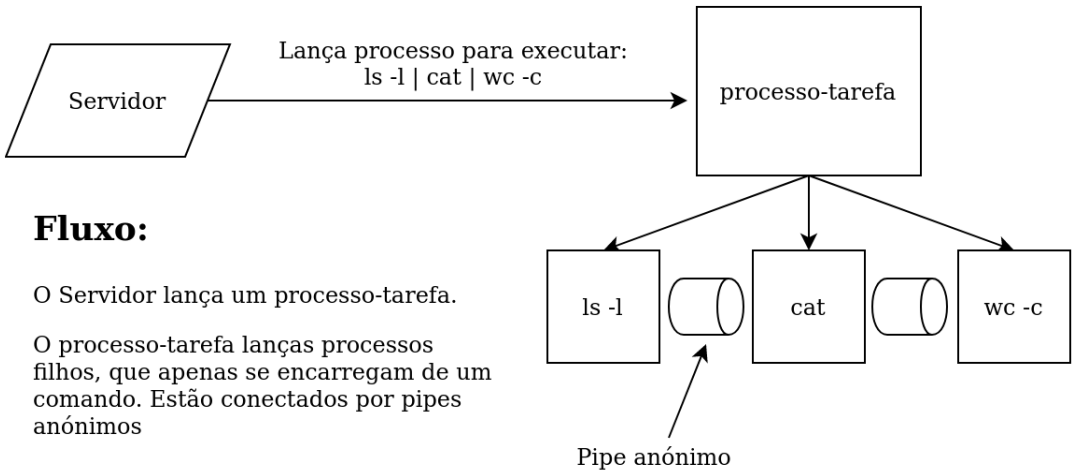

Figura 1: Lançamento de Tarefas.

[1]Explicaremos à frente que na verdade tem mais.

4.1 Tempo de Inatividade

Objetivo - O controlador deverá ser capaz de finalizar qualquer uma das suas tarefas se não se registar nenhuma atividade num dos seus pipes anónimos durante um dado período de tempo

A solução encontrada para resolver o problema do tempo de inatividade foi criar um executável que serve como processo intermédio entre dois outros processos de uma tarefa, e que está responsável por verificar o fluxo de informação entre ambos.
Este executável denominado por *checkpipe* não passa de uma cópia do comando *cat* (lê do input e escreve-o no output), com a funcionalidade adicional de que assim que é executado lança um sinal *alarm* para si mesmo com um número x de segundos. Cada vez que algo é lido como input, o alarme é re-lançado, substituindo o alarme anterior.

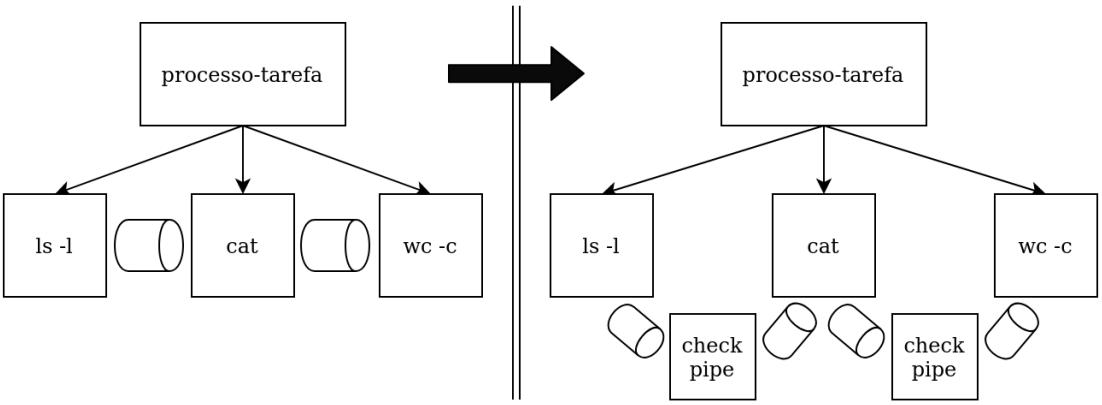

Figura 2: Utilização de um executável como camada intermédia.

Assim que o alarme dispara, o sinal *SIGUSR1* é lançado para o processo-tarefa, sinal este que como tratamento, mata todos os processos filhos criados.

4.2 Tempo de Execução

Objetivo - Restringir o tempo de execução de uma tarefa em x segundos.

Para resolver este objetivo é simplesmente lançado um alarme no processo pai que engloba todos os processos de uma tarefa (processo-tarefa) que ao ser tratado, mata todos os seus processos filhos.

4.3 Executar tarefa

Objetivo - Executar qualquer comando, como se se tratasse de uma bash.

Para se executar uma tarefa, o servidor apenas lança um processo que é nada mais do que um executável programado pelo grupo, entitulado de *run*. Este executável recebe como argumentos o comando a ser executado, o tempo máximo de execução e o tempo máximo de inatividade entre pipes (nestes dois últimos, os valor -1 corresponde ao *infinito*, i.e sem verificação). É este executável que se encarrega de criar todos os processos de um comando e redirecionar os seus *inputs* e *outputs* para pipes anonimos ligados entre si (como é de esperar, também é capaz de executar comandos que não utilizem pipes)

4.4 Listar Tarefas

Objetivo - Listar tarefas em execução.

O servidor possui em memória uma lista (implementada como uma lista-ligada) de "identificadores"de tarefas. Estes identificadores são uma estrutura de dados codificada em C como:

```
typedef struct tarefa {
    int id;
    int pid;
    int status;
    char command[MAX];
} *Tarefa;
```

Assim uma tarefa é composta pelo seu id de tarefa, o pid do seu processo (processo-tarefa), o seu *status*, i.e concluida, terminada, max inatividade, etc. E guarda também qual o comando com a qual foi chamada.

Cada vez que uma tarefa é lançada, um novo identificador é criado e adicionado à lista de tarefas em execução, desta forma listar essas tarefas é um processo simples, basta apenas percorrer essa lista em memória. A forma como as tarefas são concluídas e consequentemente excluídas desta lista é abordada mais à frente (ver 4.7).

4.5 Terminar tarefa

Objetivo - Terminar qualquer tarefa, terminando também todos os seus processos associados.

Para resolver o problema de terminar uma tarefa, todo o processo-tarefa implementa o sinal *SIGUSR2* (como é falado acima, o *SIGUSR1* já está em utilização - ver 4.1). Este sinal ao ser recebido executa um *kill* a todos os seus processos filhos, terminando a tarefa/processo em geral. Para lançar o sinal, o servidor precisa apenas de percorrer a lista de tarefas em execução (funcionalidade explicada acima - ver 4.4) e recolher o pid a que um determinado id de tarefa está adicionado (esse id é obviamente dado pelo utilizador, através do *client*)

4.6 Histórico

Objetivo - Listar registo histórico de tarefas terminadas.

Todo o histórico da aplicação (que regista todas as utilizações) é guardado num ficheiro com o nome *history*. Cada vez que uma tarefa termina, o seu identificador ,que estava guardado na lista de tarefas em execução, é retirado dessa lista e escrito no ficheiro *history* em formato binário. Assim é de notar que no histórico as tarefas aparecem por ordem de finalização e não de iniciação

4.7 Output - Funcionalidade Adicional

Objetivo - Consultar o standard output de uma tarefa já terminada.

Consultar o output de uma tarefa à escolha, foi certamente o objetivo mais desafiante do projeto. Isto deve-se ao facto de que como as tarefas correm em paralelo, saber a posição do ficheiro *log* para onde elas escreveram deixa de ser uma algo trivial. A sua conclusão foi obtida da seguinte maneira:

Primeiramente é de se notar que o servidor não fica continuamente à espera dos seus processos filhos (processos-tarefa) terminarem. Em vez disso fizemos uso do sinal *SIGCHLD*, um sinal que é lançado nos processos pais, sempre que um dos seus processos filhos termina. Este sinal é implementado no servidor, que assim que o recebe, a partir da função *wait*, faz o tratamento da tarefa terminada. O tratamento faz-se em duas partes: a 2ª parte, que tem menos urgência, é retirar a tarefa da lista de tarefas em execução e escrevê-la no histórico de tarefas terminadas. A 1ª parte trata-se de saber a posição onde a tarefa escreveu o seu output. Todas as tarefas escrevem para o ficheiro log, daqui o grupo assumiu

(uma assunção que sabemos que é perigosa) que assim que uma tarefa termina, esta terá sido a última a ter escrito algo no ficheiro log. Assim retiramos do ficheiro log a última posição escrita, e registamos no ficheiro log.idx, em formato binário, um par de dados, que contém o id da tarefa e a última posição em que ela escreveu.

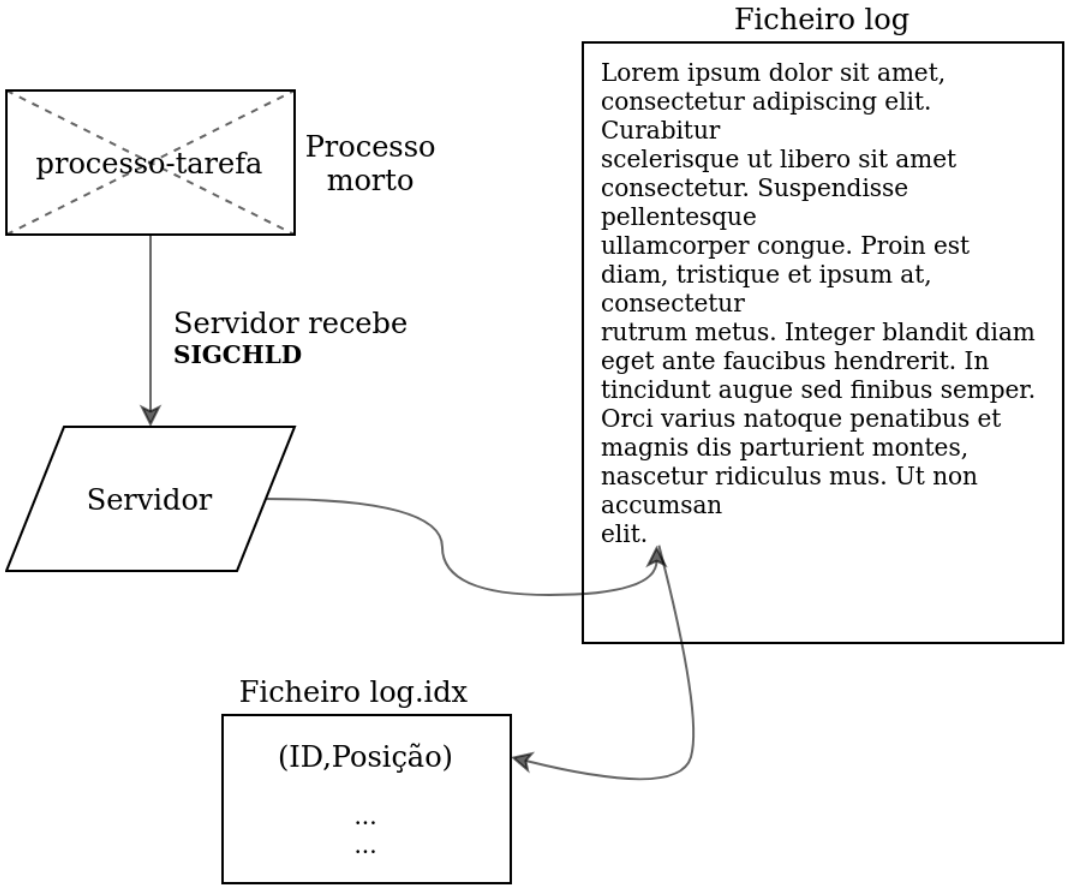

Figura 3: Processo de obtenção da última posição escrita por uma tarefa.

Desta forma, obter o output de qualquer tarefa é uma questão de procurar no ficheiro idx a posição que ela terminou de escrever, e retirar também a posição em que a tarefa antes dessa terminou de escrever, ficando assim com o início e o fim do output da tarefa pretendida.

Nota: o comando "ajuda"está também implementado e baseia-se em escritas para o output.

5 Conclusão

Podemos concluir que o grupo foi capaz de responder a todos os desafios propostos pelo guião do projeto (incluindo a implementação adicional), e por tal, capaz de conceber uma aplicação robusta para controlo e monitorização de processos. Como trabalho futuro, poderiamos melhorar o sistema de procura de endereços do ficheiro log.idx, utilizando por exemplo, uma estrutura de dados com uma procura mais eficaz (com a implementação atual a procura é linear). Estamos também cientes que o programa não é completamente *bulletproof*, i.e poderão existir certos inputs do utilizador para os quais o comportamento do programa é indefinido. A documentação do código é também outro aspeto a melhorar. Gostariamos no entanto de salientar a autonomia do programa, a criatividade do grupo, e o uso variado das técnicas de gestão e execução de processos, sinais, ficheiros e redirecionamento de escritores, dados ao longo do semestre.

Referências

[1] Standard Linux Signals
`https://www.man7.org/linux/man-pages/man7/signal.7.html`