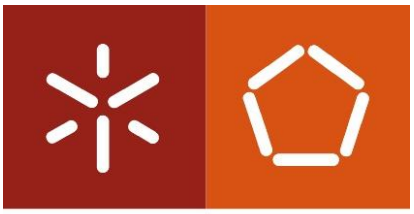

Universidade do Minho

Departamento de Informática

Controlo e Monitorização de Processos e Comunicação

Grupo Nº 21

Henrique Ribeiro(A89582)

João Correia (A84414)

Conteúdo

Capítulo 1

Introdução

As próximas páginas servirão o propósito de rever em profundidade a sofisticação da nossa implementação de um serviço de monitorização de execução e de comunicação entre processos usando conhecimentos sobre Ficheiros, Gestão de Processos, Execução de Programas, Redirecionamento de I/O, Pipes anónimos, FIFOS e Sinais.

Capítulo 2

Estrutura do trabalho

A aplicação irá adotar uma arquitetura cliente servidor. O cliente irá enviar pedidos, através de um **named pipe** para o servidor. Será depois interpretado e executado o pedido feito pelo cliente.

O servidor irá devolver informação relevante à execução para o cliente, através de um segundo named pipe. O utilizador poderá interagir com o cliente de duas formas: utilizando a linha de comandos, passando os argumentos sob a forma de flags, ou através de um próprio prompt fornecido pelo cliente.

2.1 Funcionalidades

O serviço suporta um conjunto de funcionalidades especificadas no enunciado do trabalho. Estas irão cobrir aspetos correspondentes ao controlo de execução das tarefas, assim como consultas de informação relevante à aplicação.

Algumas das funcionalidades poderão ser executadas exclusivamente no cliente, como por exemplo a opção de obter um manual de utilização, visto que não será necessário comunicar com o servidor para esse fim. No entando, a maioria das funcionalidades requer tanto um cliente como um servidor em comunicação.

Capítulo 3

Cliente

3.1 Validação das linhas de comando

O Cliente tem duas formas de ler instruções: através de um linha de comando ou através de uma interface textual interpretada (shell). Independentemente da forma como a instrução é lida, caso esta seja válida, será escrita num FIFO para comunicar com o servidor.

3.1.1 Linha de Comando

Uma vez chamado o executável do programa no terminal, a informação passada no argv[] é processada e validada. Feito este processo, é enviado um pedido para o servidor através do FIFO para ser executado o comando.

Este processo segue os seguintes passos:

- o input é validado de acordo com a flag, se esta for válida;
- os diferentes argumentos são concatenados, repondo os espaços entre eles;
- o comando é escrito no fifo para ser lido pelo servidor.

3.1.2 Shell

Na shell, o cliente escreve no stdin, a informação escrita é lida com a funcão read_line e faz-se o parsing. O primeiro componente do comando, sendo ele válido, é substituído pelo seu correspondente em flag da linha de comando, permitindo que o servidor tenha só a implementação de uma interpretação. A partir desse passo, o processo é análogo ao que foi em cima exposto.

3.2 Ajuda :: -h

Esta opção apresenta ao utilizador um manual de utilização do cliente, podendo este ser adaptado consoante o utilizador esteja a utilizar a aplicação através da linha de comandos ou a utilizar o próprio prompt oferecido pela cliente.

De forma a simplificar a utilização da aplicação, os comandos que usam flags da linha de comandos funcionam também quando se utiliza o prompt da aplicação (para além dos comandos próprios do prompt), permitindo ao utilizador apenas ter de saber utilizar as flags da linha de comandos para conseguir a utilizar a aplicação das duas formas.

Capítulo 4

Servidor

O Servidor irá ler continuamente a informação enviada pelo cliente que chega do FIFO, processando-a, executando-a. De acordo com o comando escreve uma respota direcionada ao cliente num novo FIFO.

4.1 Estrutura Record

Ao ser criada uma tarefa, é criada também uma estrutura Record associada. Esta estrutura é composta por uma string que irá guardar o conteúdo da tarefa, uma variável status, que irá guardar o estado de execução da tarefa, e o pid do processo pai que irá criar os vários processos que compõe a tarefa. Estas estruturas vão sendo guardadas num array, que será depois acedido pelas várias funcionalidades do programa.

4.2 Executar em pipe :: -e

Quando é pedido para executar mais de que um processo, são criados sucessivos forks num for loop para poder encadear os mesmos. Neste processo são usados pipes anónimos e operações de duplicação dos descritores de ficheiro para que os processos possam comunicar entre si e executar com successo a execução concorrente.

Primeiramente, é aplicada análise sintática à linha recebida pelo servidor correspondente a uma tarefa, produzindo uma matriz de strings em que cada linha corresponde a um processo e aos seus argumentos.

Sempre que se pretende executar uma nova tarefa, é criada uma instância da estrutura record que guarda, para além do nome da tarefa, o pid do processopai da mesma, e o status - inicialmente a zero - indicando o seu estado de execução.

À medida que que se dá a execução dos vários processos que compõem uma tarefa, são guardados num array os seus pids, de modo a que seja possível terminar a sua execução.

No momento da chamada do exec torna-se útil a arquitetura da matriz de strings, sendo que para cada iteração i do loop o nome do processo a executar encontra-se na coluna 0 da linha i e os restantes elementos dessa mesma linha da matriz são os argumentos correspondentes.

Quando todos os processos de uma tarefa terminam, o seu processopai terminará com um certo exit code, este corresponderá ao estado de conclusão da tarefa, indicando se terminou naturalmente ou foi interrompida devido a algumas das funcionalidades de controlo implementadas. Este exit code será apanhado num wait existente num handler específico para o SIGCHLD, que só será ativado quando o processopai da tarefa é terminado.

Na estrutura Record correspondente, é substituída depois a varíavel de status pelo status recebido no wait.

4.3 Tempo-execucao :: -m

Esta funcionalidade permite definir o tempo máximo em segundos de execução de uma tarefa.

O valor deste limite será guardado na variável global **time_limit_execute**. Quando é criado o processo-pai que irá executar os vários processos correspondentes a uma tarefa, é lançada uma chamada alarm(), com o valor deste limite. Caso este alarm chegue a ser ativado, o handler deste alarme (**timeout_handler**), irá terminar a execução de todos os processos filhos, colocando, também, o valor da variável **timeout_termination** a 1.

Esta variável permitirá ao processo verificar que ocorreu uma terminação anormal dos seus processos-filho e adaptar o seu exit code, de forma a que este processo, ao ser captado no handler do SIGCHLD, terá o exit code que corresponda à terminação por excesso de tempo, atualizando o status code na estrutura record correspondente a essa tarefa.

Esta funcionalidade é testada com a script **script_execution_limit.sh** que testa vários comandos com diferentes valores para o limite de tempo.

4.4 Tempo-inactividade :: -i

Esta funcionalidade permite definir o tempo máximo em segundos de inactividade de comunicação num pipe anónimo.

O valor deste limite será guardado na variável global **time_limit_communication**.

A verificação do tempo de inatividade dos pipes não pode ser feita ao mesmo nível da verificação do da opção -m, pois causaria conflito entre as chamadas alarm(), esta verificação ocorre dentro de cada um dos processos-filho, encarregues de executar cada um dos processos de uma tarefa.

Caso se tenha apenas um pipe anónimo a ligar dois processos, não se sabe quando estaria a ser feita a leitura. Criou-se então um segundo pipe auxiliar que irá ler do descritor de leitura do pipe que liga os processos, criando um alarme com o tempo máximo do tempo de inactividade de comunicação, que vai sendo reiniciado a cada leitura deste pipe. O que vai sendo lido é escrito no descritor de escrita deste pipe auxiliar, sendo depois o descritor de leitura deste mesmo pipe copiado para o stdin do próximo processo.

Se o alarme chegar a ser ativado, o seu handler irá enviar, ao seu processo pai, um SIGUSR2. Aqui, o handler deste sinal irá indicar-lhe que um dos seus pipes ultrapassou o limite de tempo de falta de comunicação e, como tal, todos os seus processos-filho deverão ser terminados.

A varíavel global **timeout_communication** é colocada com o valor 1, de modo a que o processopai possa lançar um exit code adaptado a esta ocorrência.

Esta funcionalidade é testada com a script **script_communication_limit.sh** que exxecuta vários comandos encadeados com diferentes valores para o limite de tempo de comunicação entre eles.

4.5 Listar :: -l

Esta opção permite listar todas as tarefas em execução no servidor.

O servidor irá percorrer a estrutura que guarda todas as estruturas Record e, sempre que um record tiver a sua variável com valor igual a 0, ou seja, indicando que o processo correspondente ainda está em execução, a informação associada a esse processo é enviada para o cliente.

4.6 Historico :: -r

Esta opção permite listar todas as tarefas cuja execução já terminou, assim como a forma como terminou.

O servidor irá percorrer a estrutura que guarda todas as estruturas Record e verifica a variável status de cada uma dessa estruturas. O valor dessa variável, em cada tarefa, é traduzido para

o modo de terminação correspondente e, a informação completa correspondente a cada tarefa é enviada para o cliente.

4.7 Terminar :: -t

Esta opção permite terminar uma tarefa que ainda esteja em execução. O processo principal do programa não tem acesso aos pids dos vários processos que compõem a execução de uma tarefa, porém o, pid do processo pai dessa tarefa encontra-se guardado na estrutura record correspondente a essa tarefa.

Para sinalizar a esse processo que deve terminar todos os seus processos-filho, é-lhe enviado um sinal SIGUSR1. O processo terá um handler associado a esse sinal (**sigusr1_handler**), que irá terminar todos os processosfilho, visto que o processo pai já tem acesso ao array de pids dessa tarefa. A variável **forced_termination** toma o valor um, de forma a sinalizar a terminação forçada. Esta variável permite ao processo pai da tarefa adaptar o seu exit code para que sinaliza ao processo principal que a terminação forçada foi concluída.

Esta funcionalidade é testada com a script **script_terminate_task.sh** que lança várias tarefas, terminando algumas que já se encontravam terminadas e outras que devem ser mesmo terminadas.

4.8 Output :: -o

Esta opção permite consultar o output gerado por uma dada tarefa. Antes da execução final de uma tarefa, o output do último exec é redirecionado para um pipe auxiliar **pipe_command_output**, este pipe é posteriormente lido e escrito e o seu conteúdo é escrito para o ficheiro **log.txt**, sendo o número de bytes escritos lidos no ficheiro **log.idx**.

Esta funcionalidade é testada com as scripts **script_output.sh**, que testa funcionalidades básicas, **script_output_terminate.sh**, que teste o output de comandos terminados pelo utilizador, **script_output_wrong_order.sh** que testa o output de tarefas que não terminam na mesma ordem com que são enviadas ao servidor e, por fim, **script_output_general.sh**, que testa todas as funcionalidades em conjunto.

4.9 Teste geral

Para além de todas as scripts de teste de cada uma das funcionalidades, foi também gerada uma script **all_features_script.sh**, que testa todas as funcionalidades disponíveis na aplicação, interligando-as entre si.

Capítulo 5

Conclusão

Em conclusão, todas as funcionalidades propostas estão implementadas de forma correta, seguindo as estipulações do enunciado e utilizando o conhecimento adquirido ao longo do semestre.

A maior dificuldade do trabalho consistiu na implementação da flag -i, que nos obrigou a alterar o paradigma de pensamento de forma a encontrar uma solução criativa que não entrasse em conflito com a flag -m.

Implementar as funcionalidades de redirecionamento de IO <e >seria um desafio que valia a pena perseguir.

Também a automatização de comandos que requerem input do utilizador quando utilizados sozinhos, como *cat* e *wc*, de forma a não interferirem com a interface do servidor, aquando da execução destes seria um aspeto a valorizar futuramente.