Sistemas Operativos

Grupo TP 109

15 de Junho de 2020

Trabalho Prático

José Ferreira (A89572) Luís Magalhães (A89528) Jaime Oliveira (A89598)

Conteúdo

1 Introdução

Este projeto foi realizado no âmbito da UC de Sistemas Operativos do 2º Ano do Mestrado Integrado em Engenharia Informática. O objetivo deste projeto passa por desenvolve um serviço de monitorização de execução e de comunicação entre processos.

O serviço permitirá a um utilizador a submissão sucessiva de tarefas, sendo estas, sequências de comandos encadeados por *pipes* anónimos. Este sistema deve suportar uma *interface* para o utilizador (*Client*) submeter as suas tarefas para serem excutadas num servidor (*Server*). o utilizador deve poder ainda terminar a execução de uma tarefa, ver a lista das tarefas em conclusão, assim como as tarefas que já terminaram (histórico), definir um tempo máximo de execução e um tempo máximo de inatividade, receber uma lista com as opções acima para saber as funcionalidades suportadas pela *interface* e por fim pedir o *output* da execução de uma tarefa em específico. A *interface* deve permitir que o cliente execute as tarefas acima descritas através de argumentos passados à execução da sua interface ou através da *prompt* "**argus$**"que lhe é apresentada quando a sua *interface* é executada sem argumentos.

2 Interface cliente/servidor

2.1 Cliente

A *interface* do servidor deve estar aberta para pudermos usufruir da *interface* do cliente. Esta, quando aberta, abre dois **FIFO**s (*pipes* com nome) entre a própria e o servidor - um para leitura (**fdr**) e outro para escrita (**fd**). É criado um processo-filho que lê o que o servidor escrever no *pipe* de leitura (**fdr**) e escreve para o *client*.

Feita a abertura dos *pipes*, vamos verificar os argumentos passados à execução do *client*.

Caso sejam passados dois argumentos (sendo que o comando '**./client**' é um deles), sabemos que apenas foi passado um argumento à execução, o que nos garante que será algo do tipo "**-x**"em que **x** será uma opção válida de execução no servidor. Neste caso, passamos exatamente esse comando ao servidor através do *pipe* de escrita.

Caso sejam passados mais do que dois argumentos, o cliente cria uma *string* em que o argumento de indice 1 - indica a tarefa a realizar - é escrito normalmente e os restantes são escritos entre dois apóstrofes, passando depois esta *string* para o *pipe* de escrita que a vai passar ao servidor.

Por último, caso o comando de execução da *interface* do cliente seja passado sem argumentos, inicia a *command prompt* da interface ("**argus$**"), vai ler do **stdin** do cliente, guardar a *string* lida num *buffer*, passá-la para o servidor através do *pipe* de escrita e repetir o processo até que seja encerrada.

2.2 Servidor

No servidor, são criados os **FIFO**s que vão ser usados para comunicar entre o servidor e o cliente. São criados exatamente 2 **FIFO**s, um para o servidor ler a informação passada pelo cliente - "**pipe_cliente_servidor**- e outro para o seridor escrever informação para ser passada para o cliente - "**pipe_servidor_cliente**". O primeiro vai ser aberto, vai ler informação e ser fechado num *loop* que impede que informação indesejada fique em *buffer*.

Abrimos também 2 ficheiros ("**log.idx**"e "**log**"), em 2 *file descriptors* (1 de leitura e 1 de escrita) para cada um dos ficheiros.

Aberto o *pipe* de leitura do servidor, vamos ler para um *buffer* a informação passada no *pipe* e analisá-la de modo a saber tratá-la para ser passada ao resto do programa.

Se no *buffer* se verificar a existência do char **' '**, sabemos que se trata de uma divisão entre um comando e os seus argumentos, então passamos para uma *string* o comando. Se o *buffer* for uma *String* de tamanho 2, sabemos que é um comando do tipo "**-x**"em que **x** é a opção que o cliente quer executar e passamos para uma *string* o próprio comando. Se a *string* lida não cair em nenhum dos casos anteriores, será simplesmente uma palavra, passada na *prompt* da *interface* do cliente e por isso devemos apenas armazená-la numa nova *string*.

Feita a análise da informação recebida, e dada à *string* "**exec**"o comando lido pelo servidor, vamos verificar em que caso, ou seja, a que opção corresponde o comando. O servidor está pronto para receber o comando "**executar**"ou "**-e**", "**listar**"ou "**-l**", "**historico**"ou "**-r**", "**tempo-execucao**"ou "**-m**", "**terminar**"ou "**-t**", "**tempo-inatividade**"ou "**-i**", "**ajuda**"e "**output**". Caso a *string* recebida não seja nenhuma destas, passa ao cliente a informação de que a opção inserida não é válida e a sugestão de o cliente inserir o comando "**ajuda**"para ver a lista de comandos disponíveis na *interface* do servidor.

O servidor tem também duas variaeis globais que são seu histórico (definido no *header* do historico) e o tempo maximo de execução, defenido por *default* a 0 mas alterável conforme a vontade do cliente.

O histórico do servidor é guardado numa Lista Ligada em que cada elemento guarda o número da tarefa, o estado da execução da tarefa, o **pid** do processo que a executa, a tarefa como uma *string* e um apontador para o próximo elemento da estrutura.

Foi desenvolvido de forma a funcionar como uma *stack* em que o elemento no topo é o último elemento a ser adicionado ao histórico.

O estado de execução pode ter 5 estados diferentes - 0 se estiver em execução, 1 se a tarefa terminou a sua execução, 2 se terminou por exceder o tempo máximo de execução do servidor, 3 se terminou por ordem do cliente e 4 se terminou por exceder o tempo máximo de inatividade.

3 Tarefas

Segue-se a explicação da execução de cada uma das tarefas disponíveis na interface.

3.1 Executar

Quando o servidor recebe o comando "**executar**"ou "**-e**", faz parse da restante informação do *buffer* para criar uma *String* "**command**"com o comando que o cliente pediu para ser executado. É criado um processo-filho que vai ser responsável pela execução do comando armazenado na *string* no qual o sinal **SIGCHLD** volta a ter o seu comportamento natural. No processo-filho criado, criamos outro processo-filho que vai executar o comando lido, passando-lhe também o **fd** de escrita dos ficheiros "**log**"e "**log.idx**"e o tempo de inatividade definido no servidor. Esta execução é feita com um tempo máximo "**timeMax**", que, se excedido, força o terminar da tarefa em execução. Se o tempo máximo estiver com o valor default 0, é ignorado pela execução do programa. Terminando a execução deste processo-filho é enviado o sinal **SIGCHILD**, que vai atualizar o estado da tarefa que foi realizada no histórico do servidor.

A execução de uma tarefa armazena num *array* de tamanho máximo 150 (valor definido com base nas previsões de limitação superior) e um valor inteiro que indica quantos **pid**s se encontram no *array*.

Passamos o **fd** do ficheiro "**log**"para o **stdin**, calculamos a posição inicial do *output* da tarefa **n** no ficheiro "**log**"e escrevemos essa posição na n-ésima linha do ficheiro "**log.idx**".

Prevê-se o uso de dois sinais - **SIGALRM** e **SIGURS1**. O primeiro usado para quando o **timeMax** tem de ser considerado e o tempo de execução da tarefa atinge esse tempo máximo. O *handler* deste sinal termina todos os processos cujo **pid** se encontra no *array* e retorna o valor 2 (relevante para o histórico e explicado adiante). O segundo é usado para forçar uma tarefa a terminar, terminando todos os **pid**s dentro do *array* da execução da própria tarefa e retornando o valor 3. Para "matar"os processos-filhos do processo de execução da tarefa, é-lhes enviado o sinal **SIGKILL**.

Quanto à execução em si, ao receber um comando, esta informação é separada para um *array* de *strings* em que cada elemento será um comando e guardamos o número total de argumentos calculado.

Criamos uma matriz com os *file descriptors* dos pipes que vão ser usados para comunicação entre os processos-filhos que executam cada comando e um *array* que guarda o valor de retorno dos processos-filhos.

Por fim, para cada comando é criado um processo-filho que vai abrir um *pipe* para ler e outro para escrever e executar o comando que lhe é atribuído. De notar que o primeiro processo-filho criado não vai ler de *pipe* nenhum e o último não vai escrever para *pipe* nenhum. O processo pai, para cada processo-filho criado, adiciona o **pid** do filho ao *array* previamente mencionado e incrementa o valor de **pid**s armazenado.

A função que executa cada comando simplesmente separa o comando dos seus argumentos e executa usando "**execvp**", que escreve o *output* no ficheiro "**log**".

O servidor atualiza o seu histórico, adicionando-lhe a tarefa que se encontra em execução através da função **addTask**.

3.2 Listar

Quando o cliente pede a listagem das tarefas em execução no servidor através do comando "**listar**"ou do argumento de execução "**-l**", vamos percorrer o histórico do servidor e criar *strings* que permitam o *output* no formato correto para o cliente das tarefas que ainda se encontram em execução, ou seja, cuja variável "**exec**"esteja a 0. Caso nenhuma tarefa se encontre em execução, essa informação também é passada ao cliente.

3.3 Terminar

Quando o cliente pretende terminar a execução de uma tarefa **n**, passa ao servidor o comando "**terminar n**"ou "**-t n**". O servidor, recebendo este comando, verifica se a tarefa n se encontra no historico e se está em execução. Se estiver em execução, vai ao histórico buscar o **pid** do processo que executa essa tarefa e envia-lhe o sinal **SIGUSR2**, cujo *handler* tratará de "matar"todos os processos-filhos e retornar o valor 3 (explicado na parte referente ao histórico). Caso a tarefa não exista no histórico ou não esteja em execução no servidor, o cliente recebe um aviso conforme a situação encontrada.

3.4 Histórico

Quando o cliente pede o historico das tarefas já terminadas no servidor através do comando "**historico**"ou do argumento de execução "**-r**", vamos percorrer o histórico do servidor e criar *strings* que permitam o *output* no formato correto para o cliente das tarefas que já terminaram, ou seja, cuja variável "**exec**"tenha um valor superior a 0. Caso nenhuma tarefa se encontre no histórico, essa informação também é passada ao cliente.

3.5 Tempo-execução

Quando o cliente pretende alterar o tempo de execução máximo de uma tarefa no servidor através do comando "**tempo-execucao**"ou "**-m**"com o valor para o qual o pretende alterar, apenas temos de alterar o valor da variável **timeMax** do servidor para o valor lido do *buffer*. É criada e enviada ao utilizador uma *string* que indica que o tempo máximo foi definido para o valor que o cliente passou ao servidor.

3.6 Tempo-inatividade

Quando o cliente pretende definir o tempo máximo de inatividade do servidor, passa ao servidor o comando "**tempo-inatividade**"com um argumento **n** que será o valor máximo de segundos de inatividade do servidor.

Aquando da execução de cada tarefa, passamos-lhe como argumento o tempo de inatividade máximo para que seja medido na execução e, caso seja excedido, envia o sinal **SIGALRM** e executa o **sigalrm_hanlder**, que termina todos os **pid**s guardados no *array* de *pid*s da execução da tarefa em questão e devolve o código de execução 4.

3.7 Ajuda

Quando o cliente passa ao servidor o comando "**ajuda**", o servidor escreve para o *pipe* que vai ser lido pelo cliente a lista dos comandos disponíveis na *interface*.

3.8 Output

Quando o cliente pede o *output* de uma determinada tarefa através do uso do comando "**output n**"em que **n** é o número da tarefa, vamos usar o *file descriptor* de leitura no servidor para o ficheiro "**log.idx**"para calcular o tamanho do *buffer* que vamos ler e posição de início de leitura no *file descriptor* de leitura do ficheiro "**log.idx**". Tendo o *buffer*, escrevemos para o **fdr** - *file descriptor* do *pipe* que serve de comunicação entre o cliente (leitura) e o servidor (escrita).

4 Conclusão

Este projeto foi um desafio interessante para pôr em prática vários conceitos abordados ao longo do semestre na UC de Sistemas Operativos.

A maior dificuldade encontrada foi organizar e trabalhar com os diferentes sinais necessários para o funcionamento correto do projeto, dado que terminar uma tarefa por exceder o tempo de inatividade ou o tempo de execução máximo, apesar dos *handlers* dos sinais serem parecidos, o código de saída teria de ser diferente. Este elevado número de *handlers* definidos ao mesmo tempo no sistema tornou algo difícil organizar e conseguir perceber o uso correto dos mesmos quando o código começou a ficar mais preenchido.

Consideramos então que conseguimos realizar o trabalho na sua totalidade, uma vez que todas as tarefas funcionam conforme o esperado, assim como a *interface* do cliente e do servidor.