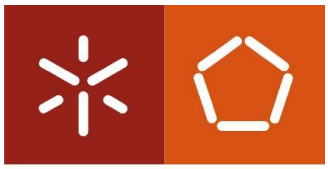


Universidade do Minho

Sistemas Operativos

Controlo e Monitorização de Processos e Comunicação

Trabalho Prático 19/20

Grupo nº98

Leonardo de Freitas Marreiros (a89537)

Conteúdo

1. Introdução

O objetivo principal deste projeto foi construir um serviço de monitorização de execução e comunicação entre processos. Para isto, foi necessário aplicar os conhecimentos sobre Sistemas Operativos adquiridos ao longo da UC, em particular a utilização de *System Calls.* O projeto engloba uma série de requerimentos que exigem o uso de diferentes fundamentos:

1. Permite uma interface *Client/Server* que possibilita a utilização concorrente de um servidor por vários clientes -> uso de FIFO'S.
2. Execução sucessiva de tarefas -> uso de *fork*, pipes anónimos e *exec*.
3. Redirecionamento de inputs/outputs -> uso de *dup.*
4. Definição de tempos de execução e inatividade -> uso de *signals* e *alarms*.

Sendo assim, este projeto colocou à prova a nossa capacidade de conseguir relacionar e mostrar destreza acerca de conhecimentos adquiridos ao longo do semestre.

Passo agora a descrever brevemente o trabalho desenvolvido, as decisões tomadas e soluções encontradas para os diferentes problemas.

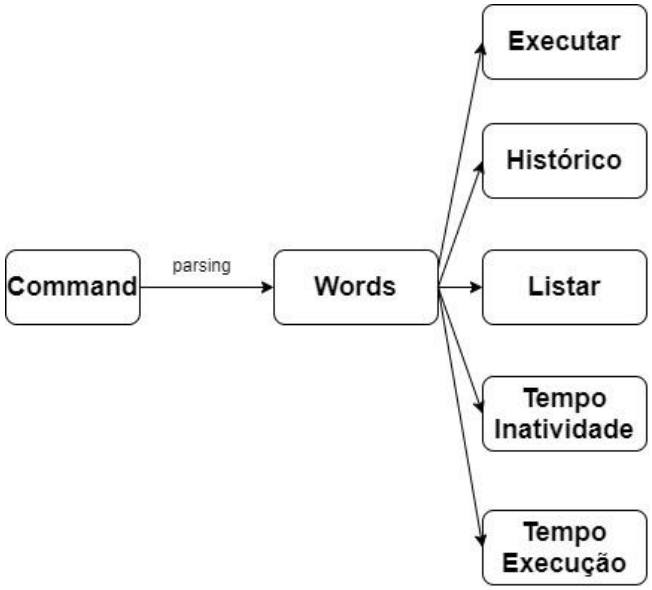
2. Divisão do Projeto

2.1 Cliente

Interface que o utilizador usa para executar comandos. Tanto pode inserir comandos diretamente a partir do terminal com os devidos argumentos, ou, caso o cliente seja invocado sem argumentos, através de uma *shell* mais *"user friendly".* É neste módulo que são lidos os diferentes comandos do utilizador e recebidas certas mensagens de confirmação. Isto é feito com recurso a dois *named pipes*: um onde são escritos os comandos que o utilizador pretende executar (neste caso, o pipe "fifo"); e o segundo, a partir do qual o cliente lê mensagens de confirmação ou outras informações vindas do servidor ("fifo2").

2.2 Servidor

No servidor são tratados os pedidos dos clientes. O Servidor recebe os comandos do utilizador, faz *parsing* destes e executa tarefas de acordo com os argumentos inseridos ou texto escrito. Além disso, como foi dito, envia mensagens de *feedback* de volta ao utilizador.



3. Análise

3.1 Executar uma tarefa

Como seria de esperar esta foi a opção mais trabalhosa pois engloba não só a execução do comando, mas também reflete as mudanças nas outras tarefas. Restringindo esta tarefa apenas à execução de comandos temos o seguinte processo:

- A parte lógica do comando inserido pelo utilizador é passada como argumento à função *runCmd* (isto é, por exemplo: "./argus -e "ls -l | wc -l"" ficaria e "ls -l | wc -l).
- Na função *runCmd* o comando é percorrido "palavra a palavra" desde o início até ao fim com diferentes comportamentos para cada palavra encontrada
- Quando um *pipe* é encontrado no comando são feitos os devidos redirecionamentos e executados os subcomandos individualmente.

3.2 Tempo de execução máximo de uma tarefa

Para esta tarefa foi criada uma variável global uma vez que este valor tem de ser respeitado para todos os próximos processos. Primeiro guardamos o valor pretendido pelo utilizador. De seguida, dentro da função runCmd é iniciado um *alarm* com este novo valor (o valor default foi escolhido como 9999 apenas como placeholder). Quando este valor é excedido, é killed o processo em execução nesse momento.

3.3 Terminar uma tarefa em execução

Terminar uma tarefa em execução exige matar todos os seus processos filho. Para isto, quando o utilizador executa um comando, são guardados num ficheiro com o nome do pid processo pai todos os pids dos processos filhos criados para executar esse comando. Quando um utilizador tenta terminar uma tarefa, apenas pode terminar aquelas que se encontram em execução. Para resolver este problema existe a função *getPids* que acede ao ficheiro das tarefas em execução e guarda o número de cada tarefa e pid correspondente. Assim, quando o utilizador insere uma tarefa válida de ser terminada, encontramos o ficheiro com o pid correspondente e é feito um kill de cada um dos pids que estão dentro desse file.

3.4 Listar tarefas em execução

Sempre que o utilizador inicia a execução de um comando é escrito no ficheiro de tarefas em execução esse comando e o seu pid associado, lógicamente, este ficheiro precisa de ser atualizado pois quando uma tarefa termina já não está, evidentemente, a executar. Para isto, sempre que um comando acaba de executar é chamada a função *updateFile* que utiliza uma série de execs para atualizar o ficheiro tendo em mente que o *file descriptor* do file das tarefas em execução não podia mudar, isto é, não podia ser eliminado e criado um novo ficheiro com o mesmo nome pois se isto acontecesse, nas próximas iterações do primeiro passo referido, estaríamos a tentar escrever para um *file descriptor* que já não existia. Sendo assim, a solução encontrada foi:

- Copiar o conteúdo do ficheiro das tarefas em execução para um novo ficheiro ("tmp.txt");
- Eliminar todo o conteúdo do ficheiro original (truncate);
- Redirecionar o output para dentro do ficheiro original;
- Executar a função unix *sed* com a keyword do pid do comando que terminou no ficheiro tmp cujo resultado será o conteúdo do ficheiro sem a ocorrência dessa "palavra";
- Eliminar o ficheiro temporário e voltar a redirecionar o output.

Desta forma atingimos o resultado pretendido.

3.5 Listar o histórico das tarefas terminadas

Uma tarefa pode terminar devido a três casos: a tarefa concluiu com sucesso; a tarefa atingiu o tempo limite de execução; e tarefa é terminada. Foram implementados diferentes métodos para atingir este resultado:

1. Tarefa terminada com sucesso: Quando uma tarefa é terminada com sucesso, a função *runCmd* devolve o valor 2;
2. Tarefa atingiu o tempo máximo de execução: Quando uma tarefa atinge o tempo máximo de execução o handler do signal SIGALRM muda o valor da variável global timeout para 1 e a função *runCmd* devolve o valor -1;
3. Tarefa terminada: Quando uma tarefa é terminada, é enviado o signal SIGUSR1 que muda o valor da variável global killed para 1.

Encontrados estes valores apenas resta filtrá-los e escrever a mensagem correspondente no ficheiro das tarefas terminadas.

4. Testes e scripts

Foram elaborados três pequenos scripts para testar as diferentes funcionalidades do projeto.

4.1 Script 1: Execução de tarefas

Para testar a execução de tarefas foi criado o programa *teste.sh* que executa algumas tarefas e depois mostra o histórico e lista das tarefas em execução.

4.2 Script 2: Tempo máximo de execução

Para testar a funcionalidade de tempo máximo de execução foi criado o programa *test2.sh* que primeiro executa um *sleep 60* e depois disso atribui o valor 10 como tempo máximo de execução e volta a executar um novo *sleep 60.* Passados 10 segundos é pedido o histórico das tarefas terminadas onde podemos verificar que a segunda tarefa já terminou enquanto que a primeira ainda está a executar.

4.3 Script 3: Terminar uma tarefa em execução

Para testar a funcionalidade de terminar uma tarefa em execução foi criado o programa *test3.sh.* Começa por executar um *sleep 60* e tenta terminar a tarefa 3, que neste caso não existe logo vai devolver uma mensagem de feedback, finalmente termina a tarefa 1 e mostra a lista de tarefas a executar e o histórico de tarefas terminadas com os resultados esperados.

5. Conclusão

O principal ponto negativo deste projeto é, obviamente, a falta da implementação da funcionalidade de tempo máximo de inatividade. Apesar da razão principal da falta desta opção ter sido a ausência de tempo, foram exploradas algumas soluções diferentes: incluir um processo a meio de cada encadeamento com um alarm e fazer reset desse alarm sempre que algo fosse lido nesses processos; tee(); flag O_ASYNC ("*Setting the O_ASYNC flag for the read end of a pipe causes a signal (SIGIO by default) to be generated when new input becomes available on the pipe.").* No entanto, mesmo sabendo que a solução iria incluir algum destes itens, não consegui concretizar e implementar no projeto. Pondo isto, um dos aspetos a melhorar seria claramente esta implementação.

Além disso, admito que as soluções encontradas para alguns dos requisitos podem não ter sido as mais corretas ou elegantes porém creio que parte deste problema se deve a ter feito este trabalho individualmente pois muitas vezes o benefício de ter colegas de grupo é, não só aliviar a carga de trabalho, mas também aconselhar e argumentar, pelo que faltou talvez essa parte de "ver soluções diferentes".

Contudo, mesmo apesar destes problemas, acredito que o projeto revela o entendimento dos conhecimentos adquiridos na UC e tem um aproveitamento bastante satisfatório e é capaz de cumprir (a maioria) dos requisitos.