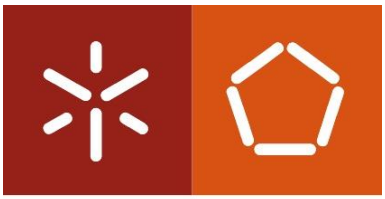


Sistemas Operativos

Grupo 43

14 de Junho de 2020

Luis Martins (A89600)

Rui Gonçalves (A89590)

Tiago Alves (A89554)

Conteúdo

1 Introdução

Este semestre, na unidade curricular de Sistemas Operativos fomos introduzidos a várias chamadas ao sistema presentes nos sistemas originados pelo Unix.

Fomos então desafiados a criar um sistema de monotorização de processos e de comunicação entre os mesmos, com essas mesma chamadas ao sistema.

Este projeto consiste em programar a interação entre um cliente e um servidor, sendo que o cliente irá escrever comandos que serão executados no servidor e cujo resultado será armazenado no ficheiro e enviado de volta ao cliente, em certos casos.

2 Cliente

No Cliente são abertos dois FIFOS previamente criados:

- Um desses FIFOS é aberto no modo de escrita (open(fifo,O_WRONLY)) para que o comando lido do standard input seja escrito no fifo de modo a ser executado pelo servidor.
- O outro FIFO é aberto no modo de leitura (open(fifo2,O_RDONLY)) de modo a ler-se do FIFO o resultado do comando executado pelo servidor e imprimi-lo no standard output.

3 Servidor

No Servidor são também abertos dois FIFOS previamente criados:

- Um dos FIFOS é aberto no modo de leitura (open(fifo,O_RDONLY)) para que o comando escrito pelo cliente possa ser executado.
- O outro FIFO é aberto no modo de escrita (open(fifo,O_WRONLY)) para que o output do comando executado possa ser enviado de volta para o cliente.

No servidor são ainda abertos dois ficheiros no modo de escrita e leitura:

- o ficheiro log contendo os outputs de todas as execuções do servidor.
- o ficheiro log.idx contendo a posição em que cada output se encontra no ficheiro logs.

3.1 Comandos implementados

3.1.1 Definir o tempo máximo de inatividade entre pipes anónimos:

O tempo de inatividade consiste no tempo que demora um processo a receber informação de um pipe anónimo.

Para a implementação desta funcionalidade, primeiro definimos uma variavél global (MAX_INATIVIDADE) que quando executarmos este comando, irá ser atualizada com o valor recebido como argumento,e esse valor ditará o tempo máximo de inatividade.

3.1.2 Definir o tempo máximo de execução das tarefas:

Para a implementação desta funcionalidade, primeiro definimos uma variavél global (MAX_EXECUCAO) que quando executarmos este comando, irá ser atualizada com o valor recebido como argumento,e esse valor ditará o tempo máximo de execução de uma tarefa.

3.1.3 Executar uma tarefa:

Para executar uma tarefa criámos um fork que irá executar uma de duas funções, uma caso o comando não tenha pipes, a *mysystem*, e outra caso o comando tenha pipes, a *mypiping*.

Na primeira, onde apenas é necessário implementar o tempo maximo de execução, antes de fazermos fork, criámos um signal handler para o SIGALRM e inicializámos um alarm com o tempo que se encontra na variável MAX_EXECUCAO. O processo pai espera que o processo filho que está a fazer o exec acabe, e se tal acontecer, "desliga"o alarm e termina com código 0, caso contrário o processo pai recebe um SIGALRM, mata o processo filho e termina com código de 1.

No segundo caso, temos o seguinte: um processo vai criar um processo filho, e este vai criar outros processos filhos para executar os diferentes argumentos dos pipes. No primeiro processo, e para o caso do processo terminar por ultrapassar o tempo máximo de execução, temos um handler para o SIGALRM que mata o seu processo filho com um SIGUSR1, e posteriormente o seu processo filho mata todos os seus filhos com SIGKILL. O processo filho tem um handler para o SIGALRM que apenas mata os seus processos filhos após não haver comunicação entre os pipes anóminos depois do tempo de inatividade definido. A forma como implementamos o tempo de inatividade foi a seguinte:

- Para o primeiro exec do pipe, após fazer fork o processo pai inicializa um alarm com o tempo de inatividade definido e espera que o seu filho acabe o processo, caso tal não aconteça o alarme "dispara"e mata o seu filho e termina com codigo 2.
- Para o caso dos pipes intermédios, é criado um novo alarme com o tempo que resta do alarm anterior, sendo o restante processo idêntico ao que foi dito anteriormente.
- Para o caso do ultimo pipe, o alarm é inicialmente desativado, pois já não vai haver mais comunicação com mais nenhum pipe, e o processo pai espera que o processo filho acabe e caso aconteça termina com código 0.

3.1.4 Terminar uma tarefa em execução:

A estratégia adotada para a implementação deste comando, consiste em, enviarmos um sinal (SIGUSR1) ao processo recebido como argumento.

A este sinal está associada uma rotina de tratamento que consiste em matar o próprio processo e todos os seus processos-filho. Todos os processos que terminam desta forma terminam com código 3.

3.1.5 Listar as tarefas em execução:

Para este comando, implementámos uma estrutura que contém todas as tarefas em execução, no momento em que o comando é invocado. Essa mesma estrutura está constantemente a ser atualizada conforme o decorrer do programa.

Quando este comando é invocado, esta mesma estrutura é percorrida e a sua informação é enviada para o cliente, em forma de String através de um FIFO.

3.1.6 Listar registo histórico de tarefas terminadas:

Para este comando, implementámos uma estrutra que contém todas as tarefas que já terminaram a sua execução, no momento em que o comando é invocado. A esta mesma estrutura são adicionadas tarefas, quando as mesmas terminam.

Quando este comando é invocado, esta mesma estrurura é percorrida e a sua informação é enviada para o cliente, em forma de String através de um FIFO.

3.1.7 Apresentar ajuda à sua utilização:

Este comando ao ser invocado, envia para o cliente na forma de String através de um FIFO, um menu com todos os comandos do programa.

3.1.8 Funcionalidade adicional:

- Consultar o standard output produzido por uma tarefa já executada:

Para este comando, guardamos no ficheiro log.idx a posição final do pointer após o output do programa ter sido escrito no ficheiro de log.

Assim sendo quando o cliente pedir o output de uma determinada tarefa, através das posições presentes no ficheiro log.idx, será possivel encontrar o respetivo output no ficheiro log, vendo a posição onde a tarefa anterior terminou, e vendo a posição onde o output pretendido terminou, lemos tudo o que está entre essas duas posições no ficheiro log, e isso será posteriormente devolvido ao cliente através do FIFO.

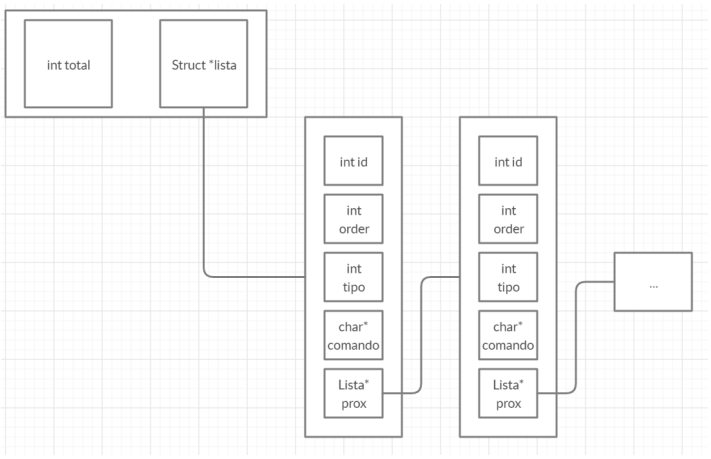
3.1.9 Teste das funcionalidades

Para testar as nossas funcionalidades, criámos um script que executa vários exemplos de comandos cada um com um resultado esperado diferente para cada uma das situações pedidas. Para uma questão de consistência, colocamos entre cada uma das execuções um sleep de 0.1 segundos.

No script existem todo o tipo de execuções: algumas que são suposto terminar, algumas que ficam a correr infinitamente caso não tenham alarm, outras que terminam por tempo de execução e outras que terminam para tempo de inatividade. Para verificar que os processos estavam a ser criados e terminados corretamente, usamos o comando *ps -a* no terminal para ver quais eram os processos que se encontravam em execução, e verificamos que antes de terminar os processos manualmente através do nosso programa encontravam-se 3 processos ainda em execução ao fim de 10 segundos(tempo de execução maximo que usamos como exemplo), e verificamos no terminal que realmente estes eram os unicos processos que ainda não tinham terminado. Depois de terminar estes 3 processos através da funcionalidade de terminação, verificamos mais uma vez no terminal que apenas 1 processo estava em execução, o servidor, ou seja o resultado esperado.

4 Estruturas de Dados

Para sabermos quais processos que acabaram ou quais os que estão em execução, armazenamos as informações sobre cada execução de um processo numa lista ligada que contém as informações sobre o pid, a ordem com que foi executada, o tipo com que acabou a execução e qual foi o comando que executou.



Quando um processo é executado, o servidor adiciona à lista do historico e à lista da execução o comando que foi escolhido pelo utilizador com tipo -1 (pois ainda não terminou) e com a ordem em que foi chamado.

Como o servidor tem de continuar a correr independentemente do processo filho que criou ter ou não acabado, não podemos fazer uma espera ativa por este processo, então para tal o servidor tem um signal handler para o SIGCHLD que deteta quando um dos seus processos filhos terminar, recolhendo assim o seu pid e o status com que terminou.

Posteriormente neste handler, vai se remover da lista de execução o processo que acabou de

terminar, vai se ver com que código terminou, e depois vai se adicionar à lista dos terminados este processo com o código que terminou.

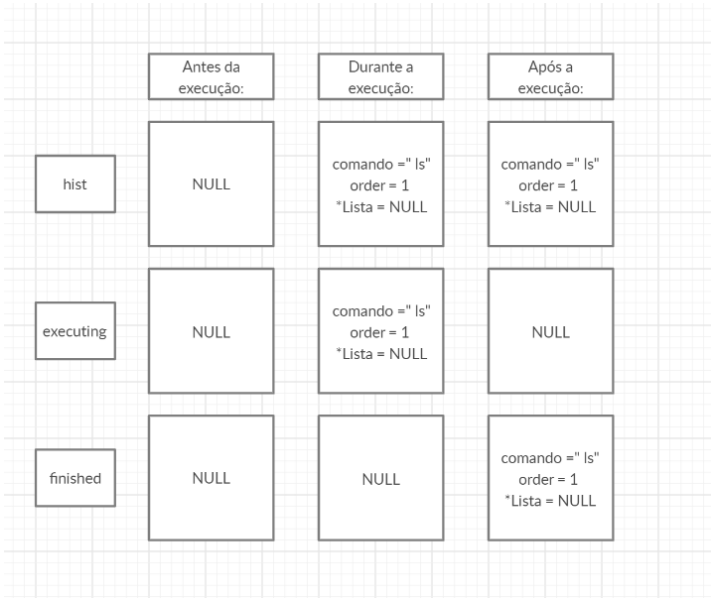


Figura 1: Conteúdo das estruturas ao longo do tempo

5 Conclusão

Cumpridos os objetivos deste projeto, poderíamos ainda ir mais além e adicionar funcionalidades como a execução de programas com redirecionamento de output para um ficheiro, por exemplo.

Pudémos observar que a programação neste nível de abstração é particularmente exigente na gestão de erros e validação dos inputs, bem como na segurança do código.

A gestão da execução de código por parte de múltiplos processos (por exemplo a hierarquia de processos presente no projeto) foi um dos principais desafios.

O desenvolvimento deste projeto ajudou-nos a desenvolver e pôr em prática os conhecimentos aquiridos durante as aulas práticas e ajudou-nos a perceber a um certo ponto como as linguagens de programação como o C têm incorporadas e sistematizadas nas suas funcionalidades o uso das system calls.