Sistemas Operativos
Grupo 54

Alexandre de Freitas Ferreira Pacheco A80760
Diogo José Cruz Sobral A82523
José Pedro Milhazes Carvalho Pinto A80741

2 de Junho de 2018

A80760

A82523

A80741

Resumo

Este semestre, na unidade curricular de Sistemas Operativos, fomos introduzidos a um conjunto de conteúdos que abordaram a maneira de funcionamento destes sistemas informáticos. Entre temas teóricos e "maneiras de pensar", fomos apresentados a um leque de *system calls* presentes nos sistemas originados pelo **UNIX**.

Neste contexto, foi-nos proposta a realização de um projeto que visa consolidar e avaliar a nossa aprendizagem sobre estes conceitos. O projeto consiste na implementação de um programa que processe *notebooks*, que são documentos de texto contendo código (a ser executado pelo programa) e output, a ser escrito pelo nosso programa nesse mesmo ficheiro.

Naturalmente, sendo um dos objetivos trabalhar a um baixo nível de abstração, o trabalho foi desenvolvido na linguagem **C** e com recurso a escassas ferramentas para além das próprias *system calls*.

1 Arquitetura do Programa

O processador de *notebooks* pode ser pensado como a junção de três "peças"fundamentais: o parser (responsável por ler o *notebook* e organizar a informação relevante numa estrutura conveniente); a execução dos comandos presentes no *notebook*; e um writer (que escreve o output gerado e organizado pelo nosso programa no ficheiro inicial). Note-se que apesar de se verificar esta estrutura funcional, não transparece, necessariamente, na organização dos módulos do código fonte.

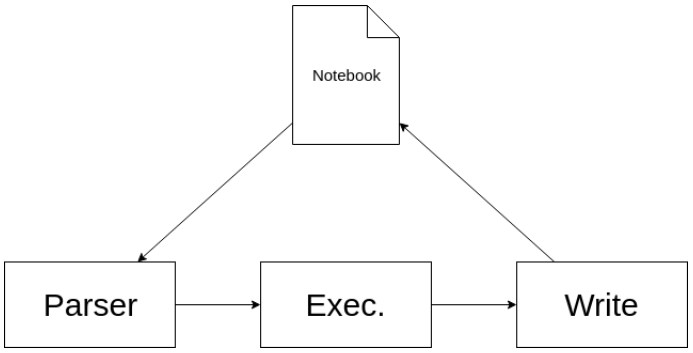

Figura 1: Estrutura básica do programa

2 Funcionalidades Implementadas

As funcionalidades deste são as seguintes:

Processamento de Comandos da bash

Quando uma linha do notebook tem o formato "`$ comando agrumento1 argumento2`", o comando nela explícito é executado. Sendo escrito a partir da linha imediatamente a seguir delimitado com as linhas "`>>>`"e "`<<<`".

Presença de texto a ser ignorado

Todo o texto que estiver em linhas não iniciadas por um cifrão ("$") é ignorado, permanecendo inalterado, ou seja, entre os mesmos comandos (inclusive os seus *outputs*).

Encadeamento de comandos

Uma linha pode ser iniciada pelo padrão "`$|`", sendo que nesse caso, será dado como input ao comando nessa linha o output do comando anterior.

Encadeamento o n-último comando

Quando uma linha começa com o padrão "`$n|`", é-lhe dado como *input* a cópia do *output* do comando "n comandos atrás". Esta funcionalidade introduz no programa uma complexidade maior, visto que o notebook deixa de ser encarado como um conjunto de "linhas de execução"e passa a ser interpretado como um conjunto de "árvores de comandos encadeados", no sentido em que o mesmo comando pode alimentar vários comandos posteriores.

Processamento de vários notebooks concorrentemente

O nosso programa pode receber vários ficheiros como argumento, processando-os concorrentemente.

3 Estruturas de Dados Utilizadas

De modo a organizar a informação convenientemente, criámos dois tipos de estruturas de dados:

1. `LCMD`, que representa sequencias de comandos numa implementação de listas ligadas.
2. `MSTRING`, que representa o *output* das diferentes linhas de código do notebook num *array* de *strings*.

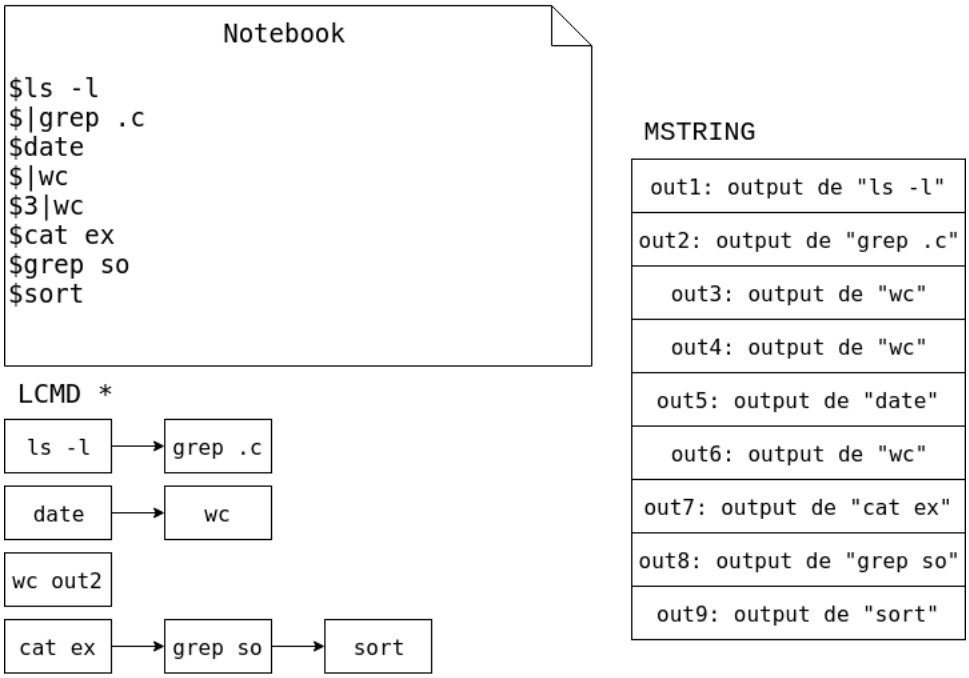

Figura 2: Estruturas de dados `LCMD *` e `MSTRING` num caso exemplo

4 Modo de Funcionamento

Quando o programa começa, é lido o *notebook*, sendo construídas na nossa estrutura de dados as sequencias de comando.

Para executar cada sequencia de comandos é feito um fork, sendo que o resultado dessa execução é no final passado por um pipe para o pai.

Na execução de cada sequência de comandos, o processo encarregue da mesma cria um filho por cada comando a executar, redirecionando os seus descritores do `stdout` para um *pipe* onde recebe o *output* de cada comando e, para além de o registar num *array* de string, o passa através do *pipe* ao processo responsável pelo comando seguinte. Quando todos os comandos dessa sequência foram executados, o processo passa ao pai todo o *output* gerado na mesma.

No caso de comandos precedidos por comandos em linhas que não a última ("$n—"), é calculada a dependência que este processo apresenta e, como o programa é sequencial, quando este comando é executado, simplesmente acede ao output do processo do qual estava dependente.

Apenas no final de todas as sequencias de comandos terem sido executadas é que o ficheiro começa a ser escrito com os *outputs* registados no *array* de strings mencionado.

5 Gestão de Erros

Tal como especificado no enunciado, no caso de um dos comandos falhar na execução, ou de o utilizador interromper o programa, este deve terminar e o notebook permanecer inalterado.

Para tal, é criado um *buffer* com um backup do ficheiro para o caso de o utilizador interromper a execução do programa, sendo este *buffer* escrito para o ficheiro. Por outro lado, caso um dos comandos não seja executado com sucesso,o programa simplesmente termina e o ficheiro não é alterado, dado que só após os comandos serem executados é que se começa a escrever os *outputs* no *notebook*.

Decidimos implementaresta solução no caso de ser o utilizador a interromper o programa dado que isto pode acontecer quando o programa já está a escrever no *notebook*.

Todos os processos que criam filhos verificam se estes conseguem executar as sua tarefas, verificando o valor passado através da *system call* `wait`, pelo que se um processonão terminar como pretendido, o programa também é terminado.

6 Conclusão

Cumpridos os objetivos deste projeto, poderíamos ainda ir mais além e adicionar funcionalidades como a execução concorrente do processamento de cada ficeiro, por exemplo.

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

A gestão da execução de código por parte de múltiplos processos (como por exemplo a hierarquia de processos presente no nosso projeto) pode-se tornar um verdadeiro desafio.

A necessidade de utilizar as *system calls* para gerir a grande maioria das funcionalidades do nosso programa faz-nos também perceber o tipo de operações e procedimentos que estão sistematizados e automatizados em linguagens de programação, ou mesmo em funções mais abstratas do próprio **C**.