Universidade do Minho
Escola de Engenharia

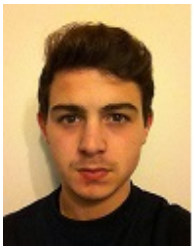
Processamento de Notebooks

Sistemas Operativos

Mestrado Integrado em Engenharia Informática
Universidade do Minho

2º Semestre
2017-2018
Grupo 3

Bruno Dantas



a74207

Daniel Rodrigues

a75655

Luís Lima

a74260

2 de Junho de 2018

Conteúdo

1 Contextualização **2**

2 Implementação **3**
2.1 Funcionamento 3
2.1.1 Visão geral 3
2.1.2 Componentes 3
2.2 Funcionalidades básicas 4
2.2.1 Execução de programas 4
2.2.2 Re-processamento de um notebook 4
2.2.3 Deteção de erros e interrupções de execução 4
2.3 Funcionalidades avançadas 4
2.3.1 Acesso a resultados de comandos anteriores 4
2.3.2 Execuções de conjuntos de comandos 5

3 Análise de resultados **6**
3.1 Testes efetuados 6
3.2 Exemplo 6

Capítulo 1

Contextualização

O projeto apresentado neste relatório surge no âmbito da Unidade Curricular de Sistemas Operativos do 2º ano do Mestrado Integrado em Engenharia Informática, com o objetivo de solidificar os conhecimentos adquiridos durante o semestre, tendo como tema principal a criação de um sistema de processamento de *Notebooks*.

A seguir estão algumas das funções importantes de um sistema operativo:

- Oferece uma descrição abstrata do hardware, permitindo um acesso unificado a ele mesmo em computadores diferentes
- Gere as tarefas que o computador deve executar, permitindo que muitos programas usem o mesmo computador, em que a cada um deles dá a impressão de que eles são a única tarefa em execução.
- Gere a comunicação local entre o computador e os seus periféricos e a comunicação mais ampla entre diferentes computadores, e até mesmo executar sistemas operativos diferentes.

Portanto, a importância de um sistema operativo é:

- Criar software independente do computador real em que está a ser executado, aumentando o número de computadores que podem usá-lo
- Oferecer um vasto conjunto de funções de software predefinidas, que de outra forma seriam reinventadas em cada programa.

Capítulo 2

Implementação

O programa foi desenvolvido para ambientes *Unix* e foi desenvolvido utilizando a linguagem de programação *C*. Ao longo deste capítulo são descritas as principais componentes do programa e as funcionalidades que este possuí.

2.1 Funcionamento

2.1.1 Visão geral

A ideia seguida para que os requisitos da aplicação sejam cumpridos (referidos no próximo capítulo) passa por guardar a informação de saída dos comandos em estruturas de dados à medida que o processamento vai sendo efetuado. Desta forma, se por algum motivo ocorrer um erro na execução de um dos comandos presentes no ficheiro *notebook*, a integridade do ficheiro original é garantida. Só no final da correta execução de todos os comandos é que a escrita sobre esse ficheiro é efetuada.

A principal estrutura de dados guarda toda a informação necessária para que seja possível controlar a execução do programa do início ao fim. Mais concretamente possuí: 1) estrutura de dados para guardar as linhas do ficheiro de entrada; 2) estrutura de dados para guardar os resultados da execução dos comandos; 3) *array* de inteiros que identificam as linhas em que os comandos estão presentes; 4) estrutura de dados para guardar as dependências entre os comandos.

A estrutura do programa principal está perfeitamente definida e consiste na execução das seguintes fases sequenciais:

1. Leitura do *notebook* com carregamento das estruturas de dados necessárias
2. Execução/processamento do notebook completando as estruturas de dados necessárias
3. Escrita da informação final sobre o *notebook*

2.1.2 Componentes

As principais componentes do programa são os dois módulos essenciais desenvolvidos, que correspondem a dois diferentes executáveis do programa. O principal - *Notebook* - é o programa principal e o outro - *Node* - é executado pelo anterior (num processo filho) para efetuar a execução dos comandos necessários.

Notebook

O programa principal possuí o controlo de toda a execução desde o início. É responsável por criar os processos filho necessários, que executam o *Node* para cada comando do *notebook* (passa como argumento a identificação dos destinos para os quais a informação de saída dos comandos deve ser escrita). Simultaneamente, efetua os redirecionamentos necessários, resultantes de dependências entre comandos do *notebook*.

Node

Este executável, invocado pelo *Notebook*, executa o comando pedido e envia o resultado para os *Nodos* que sejam dependentes destes, ou seja, para aqueles que precisem de consumir este resultado. Adicionalmente, verifica se a linha (correspondente ao comando a executar) possuí caracteres especiais. Por exemplo, caracteres que indicam redirecionamentos de/para ficheiros ('>', '<', etc), redirecionamentos para o próximo comando ('|') ou execução de comandos sem redirecionamento (';'), executando em conformidade para cada caso diferente.

2.2 Funcionalidades básicas

2.2.1 Execução de programas

A funcionalidade para execução de programas/comandos dentro do *notebook* está corretamente implementada. Dentro desta funcionalidade, destacam-se as diferentes opções de utilização possíveis: execução de apenas um comando através da utilização do caracter '$' no início da linha; receber o resultado do comando anterior como *stdin* através da utilização dos caracteres '$|' no início da linha; delimitação do resultado produzido pelos caracteres '>>>' e '<<<' (ignorados em execuções sucessivas). A explicação de como estes casos funcionam são aprofundadas nas próximas secções.

2.2.2 Re-processamento de um notebook

A possibilidade de um utilizador alterar um *notebook* já processado também é possível. Qualquer linha que não comece pelos caracteres especiais mencionados anteriormente é ignorada e vista como comentário. Desse modo, previne-se a ocorrência de qualquer tipo de erro após o re-processamento, possibilitando ainda a inclusão de novos comandos em qualquer linha do *notebook* (respeitando as regras anteriores para a execução de comandos). Como foi referido anteriormente os resultados obtidos de execuções anteriores são ignoradas, isto é, são substituídos pelos resultados obtidos na execução atual do programa.

2.2.3 Deteção de erros e interrupções de execução

O objetivo desta funcionalidade é bastante simples: sempre que um dos comandos não consiga ser executado - por ordem do utilizador ou por erro no comando - o processamento dos *notebooks* deve ser cancelado, sem efetuar alterações no *notebook* inicial.

Para cumprir este requisito, o objetivo passa por esperar que todos os comandos sejam finalizados e verificar o valor de saída de cada um destes. Se algum dos comandos falhar, ou seja, o seu valor de retorno for inferior a zero, o output não é escrito para o *notebook*. Só no final da execução de todos os comandos (a correta execução de cada um destes está assegurada) é que a escrita sobre o ficheiro é realizada.

Note-se que não é necessário tratar do sinal *SIGINT* (pressionar Ctrl+C), uma vez que não utilizamos ficheiros temporários e a escrita do *notebook* final só é efetuada se nenhum comando falhou (consideramos a escrita como uma operação atómica). Desta forma, todos os processos filho da *shell* que executou o programa principal terminam, sem comprometimento da integridade do *notebook*.

2.3 Funcionalidades avançadas

2.3.1 Acesso a resultados de comandos anteriores

De modo a garantir o acesso a resultados de um comando, optamos por criar um processo intermédio (*Node*) que se encarrega da comunicação/distribuição dos resultados e da execução do comando que lhe foi atribuído. Assim sendo, para cada comando é criado um *Node*, que o deve executar (a linha desse comando é passada nos argumentos pelo programa principal, juntamente com o número de *Nodes* que dependem dos resultados deste).

Para possibilitar o redirecionamento dos resultados entre comandos foram usados *pipes anónimos*, que são criados antes da execução do *Node* e adicionados à lista de descritores que serão herdados pelo processo filho. Após a criação do *Node*, este procede à execução do comando e prepara-se para receber os resultados deste. Assim que recebe os resultados, envia-os para o processo pai e para os *Nodes* que estão à espera dos seus resultados. Os descritores dos *pipes* são atribuídos a posições consecutivas da lista de descritores, de modo a ser usada no programa *Node*. Isto porque a primeira posição em que os descritores dos *pipes* são adicionados é predeterminada, ou seja, o *Node* só precisa de saber quantos descritores vai ter de usar. No processo de escrita dos resultados obtidos, percorre a lista de descritores e escreve-os para todos.

Para permitir a leitura de resultados de um outro *Node* usam-se os *pipes* já criados e substituí-se o descritor de leitura (*stdin*) pelo descritor de leitura do *pipe*, antes da execução do *Node*.

2.3.2 Execuções de conjuntos de comandos

Cada linha de comandos pode executar vários programas em *pipeline* com redirecionamento de dados ('|') ou sem redirecionamento de dados (;), em conjunto com a possibilidade de leitura e escrita de/para ficheiros ('>', '>','<<','2 >','2 >>'). Estas funcionalidades foram adicionadas ao programa *Node*, uma vez que é este que executa os comandos presentes na linha de comandos. O *pipeline* funciona através da execução de cada programa, redirecionando os resultados para o programa seguinte e usando *pipes* anónimos para efetuar a comunicação.

A execução sem redirecionamento permite executar vários comandos ou *pipelines* na mesma linha, garantindo que os comandos não são confundidos com argumentos de um comando anterior. A execução de comandos sem redirecionamento é feita de forma sequencial. Sempre que a sequência de comandos termina - utilização do caracter ';' - o processo espera que o comando anterior termine a sua execução, antes de proceder à execução do próximo comando (de modo a não sobrepor os resultados). É possível tornar esta execução de comandos concorrente (entre os comandos do conjunto) se não esperarmos que o comando anterior termine a sua execução. No entanto, essa opção não foi implementada.

Capítulo 3

Análise de resultados

3.1 Testes efetuados

Mediante as alterações aplicadas ao programa efetuamos testes de modo a verificar o seu correto funcionamento, sobretudo na aplicação das funcionalidade avançadas que apresentaram uma maior dificuldade de implementação e correção do seu funcionamento.

3.2 Exemplo

Para explicar melhor o funcionamento geral do programa, foi construído um diagrama que ajuda a perceber as interações entre processos durante a execução de um *notebook* de teste.

Na parte superior da imagem podemos observar, ao lado do conteúdo do *notebook*, as dependências obtidas após a leitura do mesmo (o carregamento das estruturas de dados também é efetuado nesta fase). Por exemplo, o primeiro comando **ls -f** não recebe dados de entrada (é o primeiro comando) e precisa de enviar o resultado obtido para o segundo comando **sort** e para o quarto comando **head -1**.

Em baixo, estão representados todos os processos criados ao longo da execução. Para o caso explicado em cima, podemos ver que o processo principal - *Notebook* - cria um processo filho (seta a tracejado) e executa **Node "ls -f"2**, indicando o comando e o número de *Nodes* que precisam do seu resultado. O *Node*, por sua vez, cria um novo processo filho para finalmente executar o comando **ls -f** e recebe desse filho o resultado obtido (seta **out** a preto). Após este passo, falta devolver esse resultado: 1) ao *Notebook*/processo principal (seta **out** a preto) para que adicione o resultado do comando à estrutura de dados (à qual irá aceder no final para escrever sobre o *notebook* original); 2) aos filhos dos *Nodes* que precisem do resultado para prosseguir (setas **in** representadas a vermelho).

O processo descrito em cima é efetuado para todos os comandos do *notebook*. A execução pode ser interrompida se algum dos comandos não for executado corretamente. Se todos forem executados, o *Notebook* invoca uma função para escrever corretamente sobre o antigo *notebook*. Essa função recorre às estruturas de dados auxiliares para inserir nos lugares corretos os resultados obtidos da execução dos comandos.

Processamento do notebook exemplo:

Este comando lista os ficheiros:
$ ls -f **output to: {2,4}** ~~**input form: {}**~~
Agora podemos ordenar estes ficheiros:
$| sort **output to: {3,5}** **input from: 1**
Escolher o primeiro do sort:
$| head -1 **output to: {}** **input form: 2**
Escolher o primeiro do resutado de ls:
$3| head -1 **output to: {}** **input from: 1**
Inverte o resultado do sort:
$3| sort -r ~~**output to: {}**~~ **input from: 2**

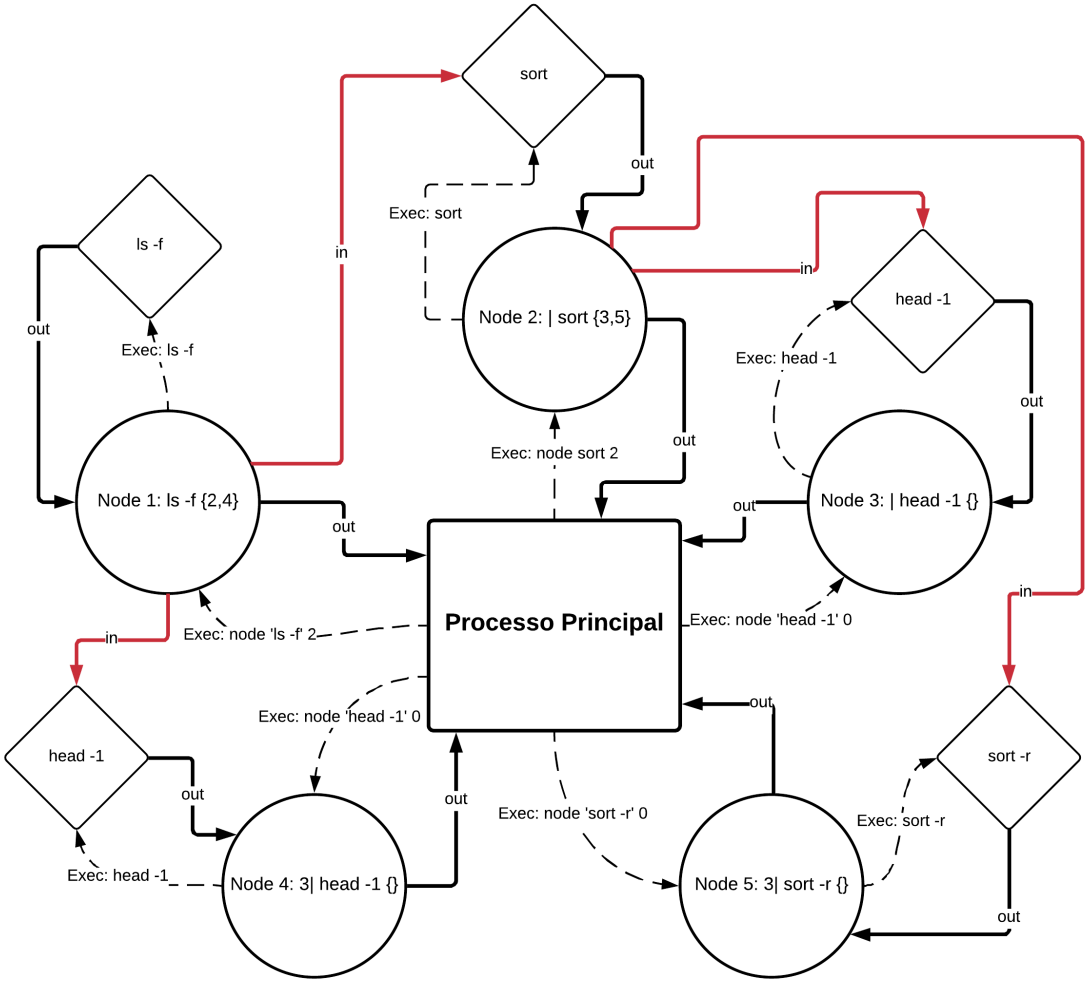


Figura 3.1: Diagrama representativo de um *notebook* exemplo