

Universidade do Minho

PROCESSAMENTO DE NOTEBOOKS

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

SISTEMAS OPERATIVOS

(2º ANO, 2º SEMESTRE, 2017/2018)

Grupo 2

A79003 Pedro Mendes Félix da Costa

A80453 Bárbara Andreia Cardoso Ferreira

A82145 Filipa Parente

Braga
Maio 2018

Índice

1	Introdução	2
2	Descrição do Problema	3
2.1	Comentários	3
2.2	Comando	3
2.2.1	Pipe para um comando anterior	3
2.2.2	Pipelining numa só linha	3
2.2.3	Execução em paralelo	3
2.2.4	Redirecionamento para ficheiros	4
3	Parsing	5
3.1	Batches	6
3.2	Ilustração da estrutura	6
4	Execução de comandos	7
4.1	Processo Pai	7
4.2	Processo Batch	7
4.2.1	Comando simples	7
4.2.2	Comando com pipelines	7
4.2.3	Dois ou mais comandos em paralelo	7
4.3	Ilustração dos processos	8
5	Opções de execução	9
5.1	<i>help</i>	9
5.2	Escrever para o <i>stdout</i>	9
5.3	Execução sequencial	9
5.4	Ler do <i>stdin</i>	9
6	Controlo de erros	10
7	Exemplo de teste	11
8	Conclusões e Trabalho Futuro	12

Introdução

Este trabalho foi realizado no âmbito da unidade curricular sistemas operativos e tem como objetivo o processamento de ficheiros de texto com comandos de bash alterando estes para incluir o *output* destes.

Este processamento foi implementando com recurso a *system calls* e à criação de múltiplos processos para que houvesse o máximo de paralelismo possível.

Descrição do Problema

Os ficheiros de texto a ser processados, chamados *notebooks*, podem conter 3 tipos de conteúdo:

- Comentários
- Comandos
- Output

Comentários

Texto simples que não é alterado de forma alguma durante o processamento. Estes não podem começar por \$, '>>>' ou '<<<'.

Comando

Linha começada por \$ que contém o comando que vai ser executado. Este tipo de linha suporta múltiplas opções adicionais.

Pipe para um comando anterior

Se, seguido do \$, estiver um número N e um | o *input* do comando será obtido do comando que estiver N posições acima no ficheiro. Caso este número seja 1 este pode ser omitido.

Pipelining numa só linha

Um comando pode ser, na verdade, um encadeamento de vários comandos unidos por pipes, por exemplo `grep -v ^# /etc/passwd | cut -f7 -d: | uniq`, neste caso o comportamento é o esperado: O *input* desta sequência poderá ser obtido de um comando anterior e o seu *output* redirecionado para um próximo comando no ficheiro.

Execução em paralelo

Numa só linha começada por \$ vários comandos podem ser executados separados por &. Neste caso o seu *output* é misturado indiscriminadamente na sua secção de *output*. Caso esta linha esteja a receber *input* de uma anterior, este *input* é distribuído para todos os comandos separados por &. Inversamente, caso algum comando esteja a receber o seu *input* desta linha, este receberá o *output* de todos os comandos presentes nesta linha também de forma "aleatória".

Redirecionamento para ficheiros

Uma linha de comando suporta também o Redirecionamento para ficheiros usando a sintaxe usual da *bash*: `<`, `>`, `>>`, `2>`, `&>`. Quando este tipo de redirecionado é utilizado, no entanto, impossibilita algumas das opções anteriores. Por exemplo, se o *output* for redirecionado para um ficheiro, nenhum dos comandos seguintes pode depender deste *output*.

Parsing

Para fazer o *parsing* do notebook foi concebida uma estrutura dinâmica que vai verificando se o ficheiro está construído corretamente enquanto organiza o seu conteúdo de forma a, posteriormente, poder ser processado mais facilmente.

A estrutura em si é um array dinâmico de **Nodes** que podem ser comentários ou comandos. Caso seja um comentário trata-se apenas de uma string.

```
struct _comment{
    String comment;
};
```

Figura 3.1: Representação em memória de um comentário

Caso seja um comando é guardada bastante informação sobre este.

```
struct _command{
    size_t dependency;
    String command;
    String output;
    IdxList dependants;
    Command pipe;
};
```

Figura 3.2: Representação em memória de um comando

A função de cada um dos campos é, respetivamente:

- **dependency**: Quantas posições atrás está o comando de cujo *output* este depende. Se este número for 0 o comando não depende.
- **command**: O comando em si. Neste ponto não é incluído o \$, o | e/ou números presentes no início da linha do notebook.
- **output**: O *output* deste comando. Inicialmente começa vazio e tem de ser mais tarde preenchido.
- **dependants**: O array das posições onde estão os comandos que dependem do *output* deste.
- **pipe**: O apontador para o próximo comando na **Batch**.

Batches

A organização dos comandos é feita sobre a forma de batches. Os comandos que estejam interligados de alguma forma estão todos colocados na mesma lista ligada, designada batch. Desta forma é possível definir blocos do notebook, que são independentes uns dos outros, para que possam ser executados de forma paralela.

Ilustração da estrutura

Dada esta organização, um exemplo de como um notebook é traduzido para esta estrutura pode ser o seguinte:

```
Comentario 1
$ ls
$| grep .c
Comentário 2
$ ps
$| wc
$2| head -2
$5| tail -2
```

Figura 3.3: Notebook exemplo com dois batches

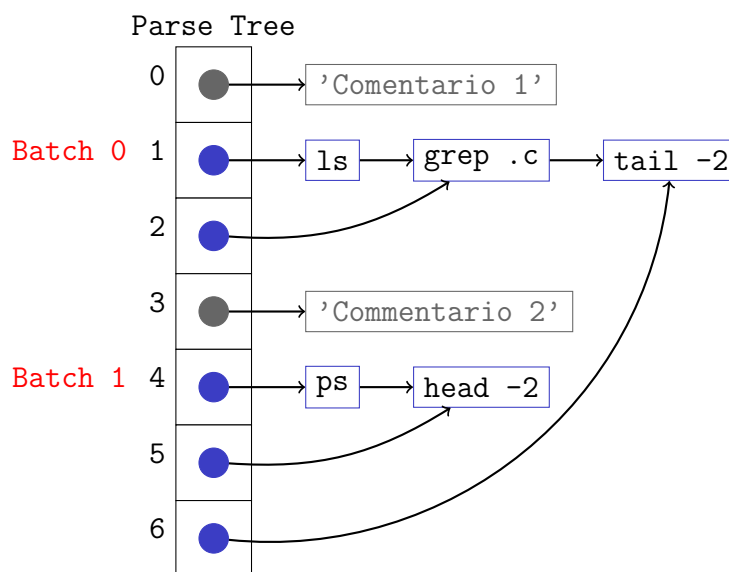


Figura 3.4: Representação em memória do notebook

Neste exemplo podemos ver que foram produzidos 2 batches. O Batch 0 é constituído pelo `ls`, `grep .c` e `tail -2`. O Batch 1 é constituído pelo `ps` e pelo `head -2`.

Execução de comandos

Processo Pai

Feito o parsing, dá-se início à execução dos comandos. Primeiro o processo pai cria um filho para cada batch e, de seguida, espera que estes morram para receber o resultado acumulado dos comandos executados por cada um através de um pipe. A partir deste resultado atualiza o *output* dos comandos e, por fim, converte a estrutura em texto que escreve no ficheiro.

Processo Batch

Cada filho irá trabalhar sobre um batch (lista ligada de comandos), criando um filho para cada nó da lista. Aqui é também decidido o método de execução de cada nodo.

Comando simples

Caso seja um comando simples é apenas redirecionado o seu *input* e *output* para pipes que comunicam com o processo encarregue do batch a que pertence. Está aqui incluído, também, o redirecionamento para ficheiros, caso seja necessário.

Comando com pipelines

Caso seja um comando com pipelines, o *input* e *output* é redirecionado de forma análoga ao caso anterior e o encadeamento é feito corretamente.

Dois ou mais comandos em paralelo

Caso o nodo contenha um ou mais comandos separados por & o tratamento é mais complexo. Primeiro, este filho cria um outro filho para cada comando que o irá executar com *stdin* e *stdout* redirecionados para um pipe cada. Enquanto os comandos executam, o pai destes (filho do batch) escreve para todos o *input* que está a receber do seu pai (batch) e agrega todo o *output* que lê dos seus filhos e envia-o, também, para o seu pai.

Ilustração dos processos

Para ilustrar este procedimento, um exemplo de como um notebook que é traduzido em processos e pipes, como descrito na secção anterior, pode ser o seguinte:

```
Batch 0
$ ls
$| grep .c
Batch 1
$ ps
$| wc & head -1
```

Figura 4.1: Notebook exemplo com um batch simples e um com comandos em paralelo

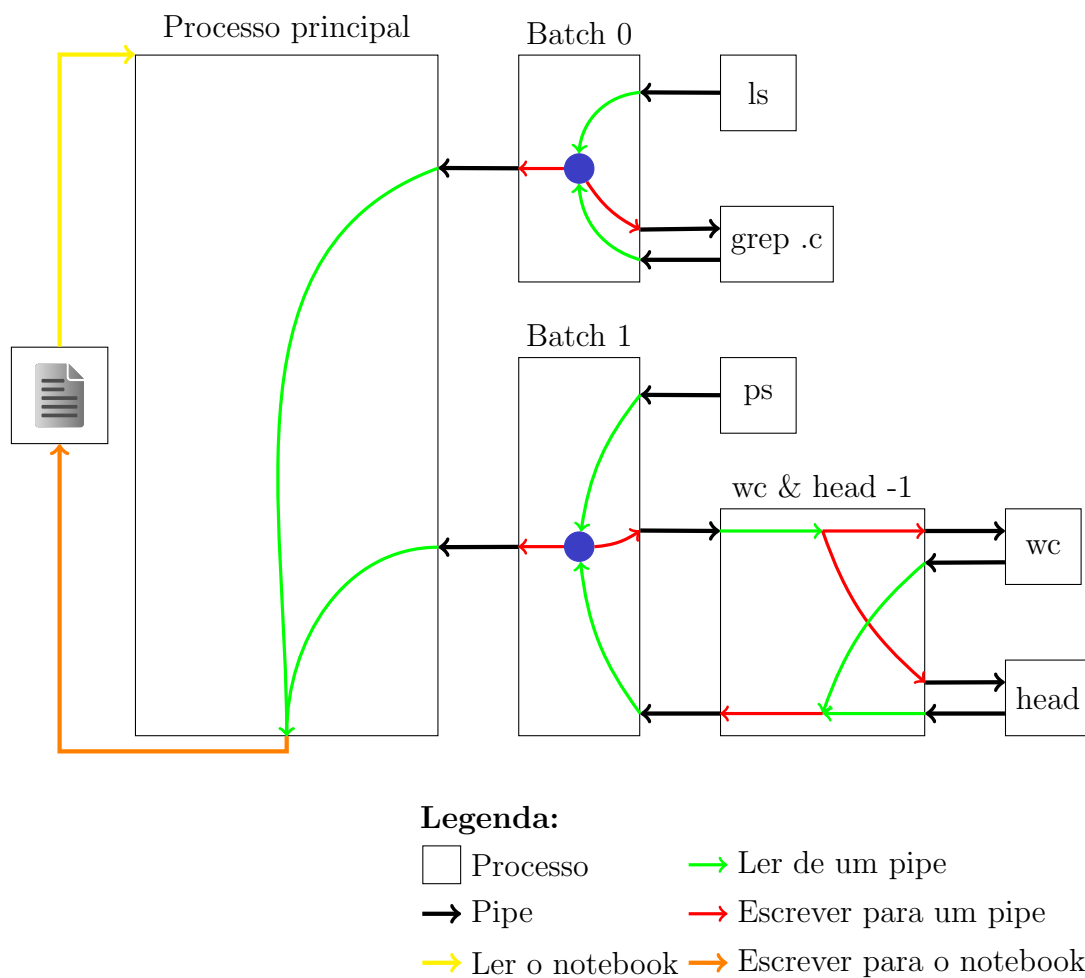


Figura 4.2: Estrutura de processos e pipes criados

Opções de execução

O comportamento do programa pode ser alterado fazendo uso de *flags* que podem ser passadas como argumento do mesmo.

help

A flag `-h` apresenta a lista de opções possíveis de passar e termina execução.

Escrever para o *stdout*

A flag `-o` faz com que o *output* produzido seja impresso para o *standard output* em vez de alterar o ficheiro de origem.

Execução sequencial

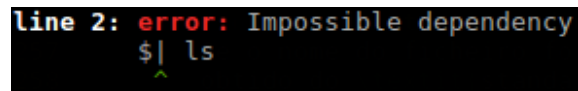
A flag `-s` faz com que as *batches* sejam executadas sequencialmente, ou seja, a *batch* *n* só é executada quando a *batch* *n-1* terminar.

Ler do *stdin*

Se o nome do ficheiro for `-` o *input* do programa será obtido do *standard input* em vez de obtido de um ficheiro. Esta opção automaticamente ativa a opção de escrever para o *stdout*.

Controlo de erros

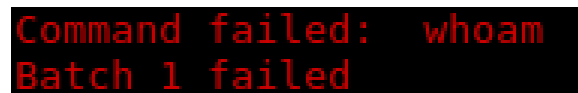
Durante a fase de *parsing* são detetados os erros de sintaxe que possam existir. Caso algum seja detetado o processamento é abortado e uma mensagem a indicar o erro que ocorreu é impressa para o *stderr*.

A terminal window with a black background. The text 'line 2: error: Impossible dependency' is displayed in red. Below it, the command '\$| ls' is shown in white. A green caret is positioned under the pipe character '|'.

```
line 2: error: Impossible dependency
$| ls
^
```

Figura 6.1: Mensagem de erro enviada caso seja pedida uma dependência impossível

Durante a execução dos comandos, se algum destes fizer o `execvp()` falhar, ou escrever para o *stderr*, a execução de todos os outros é interrompida e o ficheiro fonte permanece inalterado. A interrupção dos processos é feita da seguinte forma: Quando um comando falha este alerta o seu pai, que, por sua vez, alerta o seu pai. Este processo repete-se sucessivamente até que o topo da hierarquia seja alertado. Este sinaliza todos os seus filhos para que matem os seus filhos até que todos os processos tenham sido terminados. Atingindo este ponto o processo no topo da hierarquia termina a própria execução.

A terminal window with a black background. The text 'Command failed: whoam' is displayed in red. Below it, the text 'Batch 1 failed' is also displayed in red.

```
Command failed: whoam
Batch 1 failed
```

Figura 6.2: Erro apresentado no ecrã caso o comando não exista. Neste caso para o comando `whoam`

Exemplo de teste

Para exemplificar muitas das funcionalidades do programa foi concebido o seguinte notebook:

```
1 Comando para listar os processos
2 $ ps
3 Grep e word count desta listagem
4 $| grep notebook & wc
5 Quebrar a sequencia de comandos dependentes
6 $ whoami
7 Baralhar a listagem e escolher o primeiro elemento
8 $2| shuf | head -1
9 Ler e ordenar um ficheiro
10 $ sort < colors.h
11 Obter a ultimas 3 linhas
12 $| tail -3

Comando para listar os processos
$ ps
>>>
  PID TTY          TIME CMD
 2445 pts/18    00:00:00 zsh
15017 pts/18    00:00:00 notebook
15018 pts/18    00:00:00 notebook
15021 pts/18    00:00:00 ps
15024 pts/18    00:00:00 notebook
15025 pts/18    00:00:00 notebook <defunct>
15027 pts/18    00:00:00 shuf
15028 pts/18    00:00:00 head
15029 pts/18    00:00:00 grep
15030 pts/18    00:00:00 wc
<<<
Grep e word count desta listagem
$| grep notebook & wc
>>>
15017 pts/18    00:00:00 notebook
15018 pts/18    00:00:00 notebook
15024 pts/18    00:00:00 notebook
15025 pts/18    00:00:00 notebook <defunct>
    11      45    339
<<<
Quebrar a sequencia de comandos dependentes
$ whoami
>>>
mendes
<<<
Baralhar a listagem e escolher o primeiro elemento
$2| shuf | head -1
>>>
15024 pts/18    00:00:00 notebook
<<<
Ler e ordenar um ficheiro
$ sort < colors.h
>>>

*/
/**
#define BLUE "\033[34m"
#define BOLD "\033[1m"
#define COLORS_H
#define GREEN "\033[32m"
#define PURPLE "\033[35m"
#define RED "\033[31m"
#define RESET "\033[0m"
 * Defines bash recognized color escape sequences.
#define UNDERLINE "\033[4m"
#define YELLOW "\033[33m"
#endif /* COLORS_H */
 * \file
#ifndef COLORS_H
<<<
Obter a ultimas 3 linhas
$| tail -3
>>>
#endif /* COLORS_H */
 * \file
#ifndef COLORS_H
<<<
```

Figura 7.1: Resultado da execução do notebook

Conclusões e Trabalho Futuro

Em suma, o grupo considera que o trabalho foi realizado na sua totalidade, de forma estruturada, respondendo a todos os requisitos, e ainda adicionando requisitos não pedidos explicitamente.

Um aspeto que poderia ser melhorado era permitir que o *output* fosse redirecionado de e para ficheiros, sem comprometer as restantes funcionalidades. No entanto, esta falha pode ser contornada com as opções implementadas, apesar de ser menos eficiente.