Escola de Engenharia
Universidade do Minho

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Mestrado Integrado em Engenharia Informática
Sistemas Operativos

Processamento de NoteBooks

Grupo 46

Célia Figueiredo ae3698

Ricardo Pereira a73577

Márcia Costa a67672

Braga, 2 de Junho de 2018

Conteúdo

Resumo

O documento descreve o desenvolvimento do projeto onde foi pedida a implementação de um sistema para processamento de *notebooks*. Desta forma, um *notebook* é um ficheiro de texto que segue uma estrutura definida por descição e respetivo comando. Após análise do ficheiro de entrada, este é modificado e resulta no mesmo ficheiro alterado com os resultados da execução dos comandos no ficheiro inicial.

1. Introdução

O presente relatório documenta o trabalho prático referente a Unidade Curricular de Sistemas Operativos pertencente ao plano de estudos do 2º ano do Mestrado Integrado em Engenharia Informática. Neste projeto é pretendida a construção de um sistema para processamento de ficheiros de texto com determinadas características. Características essas que analisam pares de linhas: onde a primeira linha se refere à descrição do comando que é escrito na segunda linha iniciada com $ ou com $|. Neste último caso os comandos que aí são executados têm como entrada o resultado que o comando anterior originou.

Este documento está divido em 5 capítulos de forma a organizar a informação. No presente capítulo fazemos uma breve introdução ao projeto.

No segundo faremos uma descrição geral do projeto.

No terceiro explicaremos os ficheiros que compõem e definem a arquitetura do nosso projeto.

Relativamente ao quarto capítulo descrevemos tanto as funcionalidades básicas como as avançadas que efetivamente estão implementadas. Ainda neste capítulo expomos uma reflexão crítica onde justificamos as nossas escolhas e possiveis alternativas à nossa solução.

De forma a visualizar o funcionamento do nosso projeto, criamos o quinto capítulo onde exibimos os exemplos de teste executados.

Por fim, teremos o capítulo de conclusões e trabalho futuro que como o próprio nome indica explicamos o que fizemos e o que poderiamos ter feito, assim como os aspetos a melhorar.

2. Descrição geral do projeto

Neste trabalho foi pedido a construção de um sistema para processamento de notebooks. Neste contexto, consideramos que o *notebook* é um ficheiro de texto que após ser processado fica modificado contendo os resultados da execução dos vários comandos nele escritos.

Para finalizar o principal objetivo do projeto é o executavel receber um ficheiro de entrada com determinadas características e devolver esse mesmo ficheiro alterado com os *outputs* dos comandos executados no ficheiro de entrada, caso algo corra mal, deverá devolver um ficheiro com os erros encontrados. Como se pode observar no esquema seguinte:

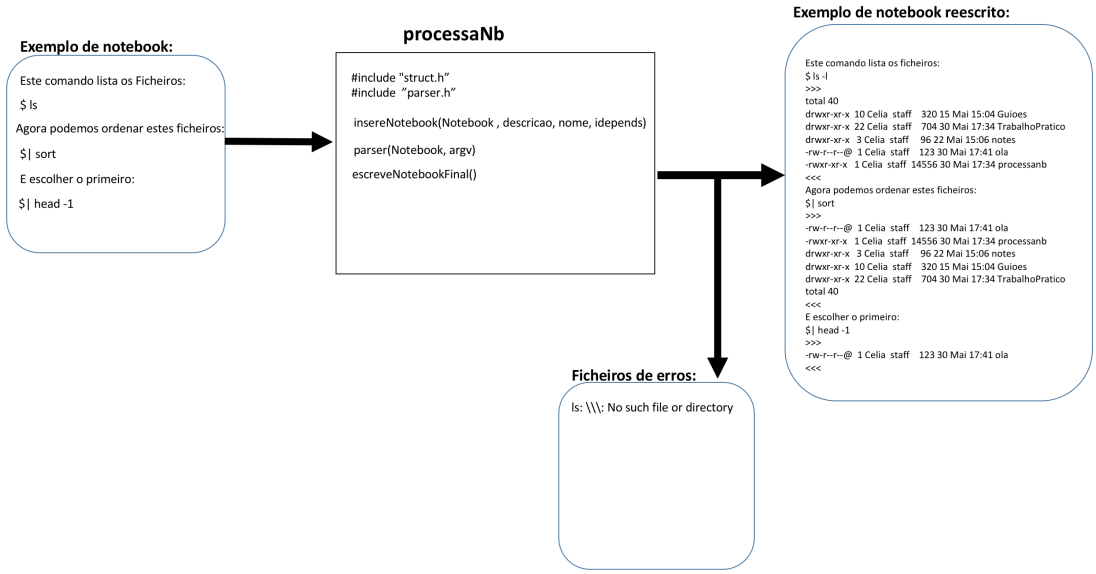

Figura 2.1: Esquema geral do projeto

3. Arquitetura do projeto

3.1 Implementação

3.1.1 main.c

O ficheiro *main* é responsável por chamar as várias funções que constituem a execução do programa principal. É neste ficheiro que são invocadas funções importantes como *parser* e *printNotebook* que estão definidas nos ficheiros *parser.c* e *struct.c* respetivamente.

3.1.2 struct.c

No ficheiro *struct.c* encontram-se definidas as estruturas *comando* e *notebook* assim como um conjunto de funções capazes de inicializar e alocar espaço para a estrutura, alterar a mesma e ainda uma função que permite imprimir a estrutura *notebook*. Basicamente aqui é feita a estruturação do projeto. Decidimos organizar o trabalho com uma estrutura de dados que guarda a informação de vários comandos, sendo estes também uma estrutura composta por descrição do comando, o nome do comando a executar, os seus argumentos, o *output* do respetivo comando e a informação relativa as dependências do comando em causa.

3.1.3 parser.c

O ficheiro *parser* está responsável por analisar o *notebook*. A análise que a função *parser* efetua, passa por verificar se o *notebook* é válido, i.e, se segue a estrutura definida no enunciado. Esta função analisa cada linha do *notebook* e para cada linha faz a respetiva avaliação. Esta avaliação vai ao encontro do que é esperado um *notebook* ser composto, ou seja, linhas começadas por caracteres diferentes de $ serão interpretadas como descrição do comando, linhas começadas por $ serão interpretadas como comandos que serão executados e linhas começadas por $ | executam comandos que têm como *stdin* o resultado relativo ao comando executado anteriormente. Há também que realçar que a função *parser* é responsável por ignorar as linhas que se encontram entre > > > e < < < , pois estas linhas referem-se aos *output*'s do processamento de um *notebook* anterior e não serão relevantes para questões de re-processamento como veremos mais à frente no presente relatório. Podemos inferir que todas as funcionalidades do projeto estão garantidas por esta função, uma vez que a mesma é responsável por efetuar os devidos redirecionamentos, a execução dos comandos (*exec*'s) no código do filho, assim como, verificar a existência de dependências e se contém ou não *pipelines* num só comando.

3.2 Makefile

Com o objetivo de automatizar tarefas para o nosso projeto definimos a seguinte *Makefile*:

```
CC=gcc
CFLAGS=-I.
DEPS = parser.h struct.h readl.h
OBJ = main.o parser.o struct.o readl.o

%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

processanb: $(OBJ)
	gcc -o $@ $^ $(CFLAGS)

clean:
	$(RM) count *.o *~
```

4. Processador de *notebooks*

4.1 Funcionalidades Básicas

4.1.1 Execução de programas

A execução do programa principal garante que existem 3 tipos de linhas a interpretar. Podemos afirmar que linhas iniciadas por um caracter diferente de $ são consideradas linhas de texto onde estão especificadas as descrições do comandos a executar, que as linhas que são iniciadas por $ são intrepretadas como comandos e nas linhas iniciadas por $| o comando tem como *stdin* o resultado do comando anterior. A função *parser* encarrega-se de ler o *notebook* de duas em duas linhas (uma descrição e um comando), analisando-as, processando-as e inserindo toda a informação resultante na estrutura. Estando o ficheiro completamente lido, é feita uma impressão da estrutura para o ficheiro *notebook*, ficando assim com os *output*'s delimitados por > > > e < < < , carateres estes que vão ser essenciais no re-processamento de um *notebook*.

4.1.2 Re-processamento de um *notebook*

Entende-se por re-processamento de um ficheiro o ato de alterar o ficheiro obtido pela primeira execução, neste contexto um utilizador poderá editar os comandos a executar e procesar novamente o *notebook*. Visualmente o programa deverá substituir os resultados anteriores pelos novos resultados. Esta funcionalidade não faz mais do que aquilo que faz um notebook sem *output*'s. A única diferença está na leitura das linhas. Quando é feita a leitura de uma linha com > > > , fazemos *readl*'s para avançar no ficheiro, ignorando o que é lido até encontrar uma linha com < < < . Quando esta linha for lida, é feita a leitura de duas novas linhas e o parser retoma o seu normal funcionamento.

4.1.3 Deteção de erros e interrupção de execução

A deteção de erros resultantes do processamento do *notebook* está, como já vimos, a ser posta em prática, principalmente na função que analisa o *notebook*. Para além daqueles casos em que o *notebook* não corresponde ao formato pedido, também existem os casos em que os *exec*'s não são executados ou então são mal executados. Nesses casos é impressa uma mensagem de erro com a função *perror*, que escreve para o *stderr* que logo no início da função *parser* é redirecionado para o ficheiro *erros.nb*. Se necessário, o utilizador poderá recorrer ao ^ C para abortar o processamento daquele comando (não é explicitado no enunciado, mas supusemos que fosse essa a intenção dos docentes).

4.2 Funcionalidade Avançadas

4.2.1 Acesso a resultados de comandos anteriores arbitrários

Para podermos aceder aos *outputs* de comandos anteriores já processados tivemos necessidade de criar a função *seeDepends* que verifica a existêcia de dependências olhando para os primeiros carateres da *string* que guarda o comando. Caso tenhamos 0 dependências significa que as linhas começadas por $ devem ser interpretadas como comandos simples e apenas fazemos o *exec* desse mesmo comando (e possíveis argumentos) no código do processo filho. Quando o número de dependências é maior ou igual a 1, significa que nos estamos a referir a comandos anteriores, o que é equivalente a ter $| (um comando) ou $n|, sendo n o número de comandos que temos que recuar para chegar àquele que tem o *output* requerido. Neste caso, criamos um segundo filho (cujo pai é o primeiro filho criado) que envia a informação do *output* do comando já processado a partir de um *pipe* para o primeiro filho (o seu pai), *pipe* esse cujo descritor de leitura é redirecionado para o *stdin*. Deste modo, o *exec*, assim que for executado, irá imediatamente retirar informação do *stdin* (caso o comando o obrigue), que é o mesmo que dizer que irá retirar informação do pipe. A partir deste ponto, o procedimento é exatamente o mesmo para qualquer caso; o pai recebe a informação do filho, isto é, o *output* resultante deste novo comando e procede à introdução deste na estrutura para que, para além de ser escrito no ficheiro final possa ser usado com comandos.

4.2.2 Execução de conjuntos de comandos

Esta foi a funcionalidade mais difícil de implemtentar pois para pensar nela foi necessário olhar para o problema com um nível de abstração muito grande! É muito fácil confundir aquilo que o pai e o filho fazem e atendendo a que iremos precisar de tantos filhos quantos os comandos encadeados, a tarefa fica ainda mais difícil. Aqui, a nossa ideia passou pela criação sequencial de filhos, isto é, o pai criava um, esse criava outro e assim sucessivamente até termos os necessários. É importante referir que a forma como criamos estes filhos é recursiva, isto é, um processo para criar um filho utiliza a função que lhe deu origem. Após isto, o último processo criado executa o *exec* e manda a informação para o pai através de um pipe, que por sua vez a utiliza no *exec* que irá executar. Assim continua até não haver mais comandos por executar. A criação de todos estes filhos e respetivos *exec*'s é feita numa função auxiliar que é chamada apenas quando este caso particular aparece e é detetado pela função *containsPipes*.

4.3 Justificação das nossas escolhas

Tendo agora a estratégia bem definida e o desenho geral do projeto, estamos já aptos para fazer um balanço deste trabalho prático. Equacionamos, no início, fazer o trabalho de uma forma totalmente diferente que utilizava unicamente ficheiros.

Não obstante, após alguma discussão apercebemo-nos de que talvez não seria sensato guardar a informação em disco, principalmente pelo facto de ser mais demorado e também pela criação de ficheiros desnecessários. Deste modo, consideravamos que esta estratégia que incluia uma estrutura era sem dúvida aquela que nos parecia mais adequada, pois através de estruturas de dados o trabalho fica mais organizado e torna-se mais fácil aceder e visualizar todos os campos que o compõem. Assim, trabalhamos essencialmente com a memória do sistema operativo, cientes de que corremos o risco de ocupar toda a memória, inclusive a *swap*, isto se um comando originar um *output* que tenha muitos *gigabytes* de informação, algo que não será muito comum.

Por outro lado também poderiamos ter um problema de bloqueio, ou seja o facto de fazermos a comunicação diretamente para o *pipe* faz com que possivelmente ocorra um bloqueio. Pois a capacidade é limitada (em geral a 4096 *bytes*) e se a escrita sobre um *pipe* continuar mesmo depois do *pipe* estar completamente cheio, ocorre uma situação de bloqueio.

4.4 Descrição de Estratégias Alternativas

Como alternativa poderíamos ter feito este trabalho utilizando apenas ficheiros, ou seja, poderíamos ter criado um ficheiro temporário que iria guardar cada linha que compõem o ficheiro *notebook* original e o respetivo *output* delimitado por > > > e < < <. Assim, no momento em que colocariamos este delimitador conseguiríamos guardar a posição atual através da *system call lseek*. O valor de retorno da*lseek* ficaria guardado num array de posições. Deste modo, quando fosse necessário aceder aos *output* dos comandos anteriores, apenas seria necessário efetuar o cálculo (o deslocamento entre a posição atual e a posição referente ao valor numérico de dependência), ler e escrever no *pipe*. No final apenas renomearíamos o nome do ficheiro temporário para o mesmo nome que o *notebook* original teria.

5. Funcionamento do Projeto

5.1 Ficheiro de input

Um *notebook* será sempre constituido por vários pares em que cada um é sempre composto por uma descrição e o respetivo comando iniciado pelo carater $. A seguir, apresentamos alguns exemplos de *notebook*'s:

```
Este comando faz uma listagem mais completa dos ficheiros:
$ ls -l
Agora podemos ordenar estes ficheiros:
$| sort
E escolher o primeiro:
$| head -1
```

Exemplo em que usámos o segundo comando anterior.

```
Este comando lista os ficheiros:
$ ls
Agora podemos ver a data atual:
$ date
E escolher o primeiro componente do segundo comando anterior:
$2| head -1
```

Exemplo em que usámos um comando que lista muita informação.

```
Este comando lista os ficheiros:
$ ls -lR /
Agora podemos ver a data atual:
$ date
E escolher o primeiro componente do segundo comando anterior:
$2| head -1
```

5.2 Output

Uma vez processado, um *notebook* também terá sempre uma forma específica; a descrição, o comando e o resultado do comando delimitado por > > > e < < < . A seguir, apresentamos alguns exemplos de *notebook*'s processados, respetivamente:

Este comando faz uma listagem mais completa dos ficheiros:

```
$ ls -l
>>>
total 4672
-rw-r--r--   1 Celia  staff      217  1 Jun 15:32 Makefile
drwxr-xr-x  16 Celia  staff      512  1 Jun 15:31 Relatorio
-rw-r--r--   1 Celia  staff  2277376  1 Jun 15:32 core
-rw-r--r--@  1 Celia  staff        0  1 Jun 19:44 erros.nb
-rw-r--r--   1 Celia  staff     1017  1 Jun 15:32 main.c
-rw-r--r--   1 Celia  staff     3648  1 Jun 19:19 main.o
-rw-r--r--@  1 Celia  staff      123  1 Jun 19:25 notebook.nb
-rw-r--r--@  1 Celia  staff     2173  1 Jun 19:44 ola
-rw-r--r--@  1 Celia  staff     4968  1 Jun 19:18 parser.c
-rw-r--r--   1 Celia  staff      175 21 Mai 14:51 parser.h
-rw-r--r--   1 Celia  staff     8680  1 Jun 19:19 parser.o
-rwxr-xr-x   1 Celia  staff    17012  1 Jun 19:19 processanb
-rwxr-xr-x   1 Celia  staff     8456 21 Mai 14:51 readl
-rw-r--r--@  1 Celia  staff     1011  1 Jun 19:18 readl.c
-rw-r--r--   1 Celia  staff      158  1 Jun 19:18 readl.h
-rw-r--r--   1 Celia  staff     3276  1 Jun 19:19 readl.o
-rw-r--r--   1 Celia  staff     4859  1 Jun 15:32 struct.c
-rw-r--r--   1 Celia  staff      802 21 Mai 14:51 struct.h
-rw-r--r--   1 Celia  staff    10412  1 Jun 19:19 struct.o
<<<
```

Agora podemos ordenar estes ficheiros:

```
$| sort
>>>
-rw-r--r--   1 Celia  staff      158  1 Jun 19:18 readl.h
-rw-r--r--   1 Celia  staff      175 21 Mai 14:51 parser.h
-rw-r--r--   1 Celia  staff      217  1 Jun 15:32 Makefile
-rw-r--r--   1 Celia  staff      802 21 Mai 14:51 struct.h
-rw-r--r--   1 Celia  staff     1017  1 Jun 15:32 main.c
-rw-r--r--   1 Celia  staff     3276  1 Jun 19:19 readl.o
-rw-r--r--   1 Celia  staff     3648  1 Jun 19:19 main.o
-rw-r--r--   1 Celia  staff     4859  1 Jun 15:32 struct.c
-rw-r--r--   1 Celia  staff     8680  1 Jun 19:19 parser.o
-rw-r--r--   1 Celia  staff    10412  1 Jun 19:19 struct.o
-rw-r--r--   1 Celia  staff  2277376  1 Jun 15:32 core
-rw-r--r--@  1 Celia  staff        0  1 Jun 19:44 erros.nb
-rw-r--r--@  1 Celia  staff      123  1 Jun 19:25 notebook.nb
-rw-r--r--@  1 Celia  staff     1011  1 Jun 19:18 readl.c
-rw-r--r--@  1 Celia  staff     2173  1 Jun 19:44 ola
-rw-r--r--@  1 Celia  staff     4968  1 Jun 19:18 parser.c
-rwxr-xr-x   1 Celia  staff     8456 21 Mai 14:51 readl
-rwxr-xr-x   1 Celia  staff    17012  1 Jun 19:19 processanb
drwxr-xr-x  16 Celia  staff      512  1 Jun 15:31 Relatorio
total 4672
<<<
```

```
E escolher o primeiro:
$| head -1
>>>
-rw-r--r--   1 Celia  staff      158  1 Jun 19:18 readl.h
<<<

Este comando lista os ficheiros:
$ ls
>>>
main.c
main.o
Makefile
notebook1.nb
notebook.nb
parser.c
parser.h
parser.o
processanb
readl
readl.c
readl.h
readl.o
Relatorio
struct.c
struct.h
struct.o
<<<
Agora podemos ver a data atual:
$ date
>>>
qua mai 30 23:55:43 WEST 2018
<<<
E escolher o primeiro componente do segundo comando anterior:
$2| head -1
>>>
main.c
<<<
```

6. Conclusões

Com a realização deste trabalho concluímos que não tomamos a melhor decisão do ponto de vista de gestão de memória, pois caso um dos comandos liste uma quantidade extremamente grande de carateres, na ordem das dezenas de gigas, tanto a memória RAM como a SWAP, muito provavelmente iriam ficar completamente cheias impossibilitando o normal processamento do programa. Contudo o programa também tem algumas vantagens em relação à escrita em disco, que é a rapidez. Um programa que escreva em memória é seguramente mais rápido que um programa que escreva para disco, e do ponto de vista de organização do código é muito melhor, pois para aceder à estrutura basta invocar os seus campos.

6.1 Trabalho Futuro

A longo prazo poderíamos acrescentar *pipelines*, funcionalidade que está perto de ser concluida, faltando apenas a comunicação dos filhos para o pai dos resultados dos *exec*'s. Poderíamos também acrescentar programação concorrente, que a nossa ver passaria por por a executar paralelamente comandos que não têm dependências.