Universidade do Minho

Mestrado Integrado de Engenharia Informática

Trabalho Prático de Sistemas Operativos

Autores:

Bruno Ferreira	A74155
José Bastos	A74696
Sara Pereira	A73700

3 de Junho de 2018

Conteúdo

1 Introdução

Este projeto foi-nos proposto no âmbito da UC Sistemas Operativos e tinha como objetivo a consolidação de conhecimentos adquiridos nas aulas práticas e teóricas. Desta maneira, o trabalho consistia no desenvolvimento de um processador de notebooks que fosse capaz de interpretar ficheiros, executar os comandos contidos pelo mesmo, incluindo comandos embebidos, e, de seguida, acrescentar os resultados dos comandos ao ficheiro que foi interpretado.

2 Arquitetura da solução

Após a análise do problema proposto, foram pensados 5 módulos, correspondentes às diferentes partes do problema.

Foi criado o módulo **Processor**, que trata da lógica de execução do programa. Para poder guardar informações dos ficheiros a serem processados, e fazer a gestão da estrutura que os guarda, foi criado o módulo **Structure**. Foi também verificado que se poderia reutilizar o código feito anteriormente num dos guiões da UC, sendo o mesmo guardado no módulo **Buffer**. Este permite fazer a leitura de uma linha do ficheiro, utilizando uma estrutura auxiliar. O módulo **Structure** é depois usado para fazer o processamento do ficheiro, guardando as informações necessárias do ficheiro na estrutura. Este processo é feito no módulo **Compiling**. O ultimo módulo criado é necessário para a execução do programa, tendo o nome de **Executing**. Neste também é tratada a escrita do ficheiro após toda a execução do programa.

2.1 Processor

É neste módulo que se encontra a main do programa, onde é tratada a lógica de execução do programa. Inicialmente é criado a estrutura Notebook (definida posteriormente) e preenchida a estrutura com as informações do notebook a ser processado. Após este processo é necessário fechar o descritor do ficheiro, que se encontra aberto apenas para leitura. De seguida é pedida a execução do notebook, e realizada a reescrita do ficheiro. Existe a possibilidade do processamento de múltiplos ficheiros, passados como argumento na execução do programa. A execução dos notebooks é concorrente, ou seja, é criado um processo para o processamento e execução de cada notebook. Isto permite que o tempo de execução do programa seja limitado pelo maior tempo de execução entre todos os notebooks.

2.2 Structure

Este módulo implementa toda a estrutura de um Notebook, que, basicamente, é composta por um array de **Commands** e respetivos dados. Primeiramente será analisada a estrutura **Command** e, seguidamente a estrutura Notebook:

Começando por analisar a estrutura de um comando **Command**:

```
typedef struct command{
    char** lines_before;
    int l_max;
    int l_num;
    int dep;
    char* command_line;
    char** command;
    char** output_lines;
    int o_max;
    int o_num;
}* Command;
```

Um comando é composto por um array dinâmico que guarda todas as linhas que o antecedem (**lines_before**), assim como o número máximo de linhas anteriores (**l_max**) e o número atual dessas mesmas linhas. Também é guardada, se existir, a sua dependência para com outro comando (**dep**), a linha correspondente ao comando atual, um array que guarda os comandos devidamente preparados para serem executados pelo **execvp** e o array correspondente ao respetivo output. Por último, o maior número possível de outputs, assim como o número atual são componentes desta estrutura.

Em relação à Notebook:

```
typedef struct notebook{
    Command* commands;
    char* filename;
    int command_max;
    int command_size;
    Buffer file;
    int rollback;
}* Notebook;
```

Como foi referido, esta estrutura guarda um array de comandos (**commands**), o nome do ficheiro a ser interpetado (**filename**), o número máximo de comandos que este contém (**command_max**) em conjunto com o número atual (**command_size**). Ao mesmo tempo, é necessária a estrutura auxiliar de leitura do ficheiro (**file**) e a variável que indica a existência de *rollback* ou não. É também definida a função que permite imprimir um dado notebook para o stdout.

2.3 Buffer

Neste módulo é criada uma estrutura auxiliar para a informação do ficheiro a ser interpretado, sendo esta inspirada num dos exercícios do primeiro guião das aulas práticas. Assim, a estrutura é composta da seguinte maneira:

```
    struct buffer_t{
    char* buf;
    int fildes;
    int nbytes;
    int index;
    int lidas;
};
```

Onde **buf** é o array para onde vai ser guardado o que for lido no ficheiro, 1024 bytes de cada vez. A variável **fildes** corresponde ao descritor do ficheiro a ser lido, **nbytes** corresponde ao seu tamanho e **index** corresponde à posição na qual o buf será preenchido. O número de carateres lidos pela função *read* será guardado em **lidas**. É, também, neste módulo definida a função de leitura de um ficheiro linha a linha cujos parâmetros são a estrutura que guarda informação sobre o ficheiro a ser interpretado (**Buffer buffer**), o array onde será guardado tudo o que é lido pela função *read* (char* buf) e o número máximo de carateres existentes no **buf**. A sua implementação começa por verificar se o buffer existe, caso contrário é chamada a função handler de erros (*perror("Null buffer")*. De seguida, o ficheiro começa a ser lido, sendo sempre verificado se houve erro ou não. Caso o número de carateres que foi lido seja menor ou igual a 0 é quebrado o ciclo.

2.4 Compiling

Neste módulo faz-se a interpretação do ficheiro, guardando as informações que se pretende guardar. Inicialmente, na função de **populate_notebook** cria-se a instância de um comando, com a função **create_command** a estrutura **commands** que está dentro da própria estrutura do notebook. De seguida, vai lendo as linhas do ficheiro contido no notebook (guardando-as como descrição do comando), até encontrar uma linha que começe por $, nesse caso adiciona ao comando, passando depois para o próximo comando. Se for necessário duplica-se o número máximo de comandos. No caso de reconhecer >, devido ao reprocessamento, não são guardadas as linhas até ao caso de aparecer < no inicio da linha, sinal que o output do comando anterior acabou. Existem também funções, como **update_lines** e **update_command** para atualizar a informação que está guardada na estrutura, funções para verificar se existem dependências no comando pretendido (**verify_dependencies**) e ainda uma função, **split_command_line** que pega na linha inteira do comando e separa o comando propriamente dito dos argumentos do comando.

2.5 Executing

Neste módulo trata-se da execução dos comandos presentes no notebook. Para a sua execução dá-se uso de **pipes**, de forma a que haja uma comunicação entre os processos dos comandos.

Para realizar a execução dos comandos percorremos os mesmos, que se encontram guardados na estrutura, executando um de cada vez. É aqui verificado se as dependências anteriormente definidas estão dentro dos limites do array presente. É feita a chamada de uma função **exec_cmd** que trata da lógica de execução de um comando. Na função de execução de comandos o processo filho fecha a escrita do **pipe_in**, duplica a leitura do **pipe_in** como standard input, caso esse comando tenha dependências, e fecha a entrada do **pipe_in**. De seguida faz o mesmo para o **pipe_out**. Já o processo pai fecha a entrada do **pipe_in**, e escreve as linhas do output na escrita do **pipe_in**, caso seja necessário, fechando-o quando estiver concluído. Espera pela conclusão do processo filho e de seguida, fecha a escrita do **pipe_out** e vai lendo o que está na leitura do **pipe_out**, atualizando assim o ficheiro de input, reescrevendo nesse mesmo ficheiro, adicionando o output gerado. No final, fecha a leitura do **pipe_out**. No caso de ocorrerem erros na execução do comando, é imprimido o ficheiro que contem o erro e o local.

3 Testes e resultados

Após a criação do programa, foram realizados alguns testes. Os ficheiros de teste encontram-se em anexo.

Execução de um teste

```
→ make
mkdir -p obj
gcc -Wall -std=c11 -o obj/structure.o -c src/structure.c
gcc -Wall -std=c11 -o obj/processor.o -c src/processor.c
gcc -Wall -std=c11 -o obj/compiling.o -c src/compiling.c
gcc -Wall -std=c11 -o obj/executing.o -c src/executing.c
gcc -Wall -std=c11 -o obj/buffer.o -c src/buffer.c
gcc -Wall -std=c11   -o notebook  obj/structure.o  obj/processor.o  obj/compiling.o  obj/executing.o  obj/buffer.o
To execute, run ./notebook <filenames>
⚡ Eletro @ ~/Projetos/SO_18 (master ✖)
→ ./notebook testes/1.txt
Foi criado o processo 28009, para executar o notebook "testes/1.txt"
⚡ Eletro @ ~/Projetos/SO_18 (master ✖)
→ cat testes/1.txt
Este comando lista ficheiros:
$ ls
>>>
enunciado-so-2017-18.pdf
Makefile
notebook
obj
README.md
src
testes
<<<
Agora podemos ordenar os ficheiros:
$| sort
>>>
enunciado-so-2017-18.pdf
Makefile
notebook
obj
README.md
src
testes
<<<
E escolher o primeiro:
$| head -1
>>>
enunciado-so-2017-18.pdf
<<<
```

Execução de multiplos testes

```
⚡ Eletro @ ~/Projetos/SO_18 (master ✖)
→ ./notebook testes/1.txt testes/2.txt
Foi criado o processo 4146, para executar o notebook "testes/1.txt"
Foi criado o processo 4147, para executar o notebook "testes/2.txt"
⚡ Eletro @ ~/Projetos/SO_18 (master ✖)
→
```

Reprocessamento de ficheiros

```
⚡ Eletro @ ~/Projetos/SO_18 (master ✖)
→ ./notebook testes/1.txt
Foi criado o processo 4945, para executar o notebook "testes/1.txt"
⚡ Eletro @ ~/Projetos/SO_18 (master ✖)
→ ./notebook testes/1.txt
Foi criado o processo 4983, para executar o notebook "testes/1.txt"
⚡ Eletro @ ~/Projetos/SO_18 (master ✖)
→ cat testes/1.txt
Este comando lista ficheiros:
$ ls
>>>
enunciado-so-2017-18.pdf
Makefile
notebook
obj
README.md
src
testes
<<<
Agora podemos ordenar os ficheiros:
$| sort
>>>
enunciado-so-2017-18.pdf
Makefile
notebook
obj
README.md
src
testes
<<<
E escolher o primeiro:
$| head -1
>>>
enunciado-so-2017-18.pdf
<<<
```

Caso de erro.

```
⚡ Eletro @ ~/Projetos/SO_18 (master ✖)
→ ./notebook testes/1.txt testes/3.txt
Foi criado o processo 4611, para executar o notebook "testes/1.txt"
Foi criado o processo 4612, para executar o notebook "testes/3.txt"
Erro no exec: No such file or directory
Command $| sorte

Erro no ficheiro testes/3.txt
⚡ Eletro @ ~/Projetos/SO_18 (master ✖)
→
```

4 Conclusão

Este trabalho, que abrange a matéria abordada na unidade curricular, exigiu um espírito de autonomia ao grupo pois ao longo do seu desenvolvimento deparámo-nos com algumas dificuldades, que foram ultrapassadas com sucesso

devido às soluções apresentadas ao longo do trabalho.

Em jeito de conclusão, considerámos que os objetivos do projeto foram alcançados, tendo o grupo feito um bom trabalho no desenvolvimento do mesmo. Julgámos também que foi um trabalho bastante enriquecedor, na medida em que fortalecemos alguns dos conhecimentos que englobam a matéria de Sistemas Operativos.

5 Anexos

Em anexo seguem-se alguns ficheiros de teste.

5.1 1.txt

```
Este comando lista ficheiros:
$ ls
Agora podemos ordenar os ficheiros:
$| sort
E escolher o primeiro:
$| head -1
```

5.2 2.txt

```
    Testar comandos sem dependências:
$ ls
Outro comando sem dependências:
$ pwd
```

5.3 3.txt

```
    Este comando lista os ficheiros:
$ ls
Agora podemos ordenar os ficheiros:
$| sorte
E escolher o primeiro:
$| head -1
```

5.4 4.txt

```
    Vamos testar comandos com dependencias maiores que 1
E com várias linhas de descrição
$ ls
Agora fazemos mais um comando aleatório:
$ pwd
```

```
$2| sort
E tentamos o ls outra vez:
$3| head -2
Mais um comando para verificar a execução:
$ ps
```

5.5 5.txt

```
    Vamos testar mais de 10 comandos antes
$ ls
Adicionamos um comando extra
$ pwd
Mais um comando sem dependências
$ ps
Tendo mais comandos:
$ date
Sendo necessário 10 comandos:
$ uname -a
$ uname
$ ls -l
$ python --version
$ ls -a /etc/passwd
Comandos sem output:
$ sleep 2
Comando com dependência maior de 10
$10| head -2
```