Universidade do MInho
Mestrado Integrado em Engenharia Informática
Junho 2018

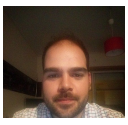
Sistemas Operativos

Processador de Notebooks

GRUPO

Tiago Baptista - 75328 

Ricardo Canela - 74568 

Lucas Pereira - 68547 

Indice

1 - Introdução

No âmbito da cadeira de Sistemas Operativos foi proposto a elaboração de um processador de ficheiros de texto, de maneira a executar comandos nele presentes e acrescentar os outputs destes comandos ao ficheiro inicial.

O programa terá de executar os comandos presentes no notebook, que nada mais é que um ficheiro de texto em que as linhas começadas por $ representam um comando, sendo que o resultado vem escrito logo de seguida e delimitado por >>> e <<< e as linhas começadas por $ | requerem que o comando a executar tenha como input o output de um comando anterior.

Para além destas funcionalidades básicas foram dados como funcionalidades avançadas, como o acesso a resultados de comandos anteriores arbitrários e a execução de conjuntos de comandos.

2 - Estrutura da Aplicação

A aplicação é constituída por um ficheiro principal main onde é chamada a **load_cmd_array** que permite alocar memória e criar uma estrutura do tipo **array_data** que nada mais é do que uma array dinâmico.

```
struct array_data {
    Command *pointer;
    int counter;
    int size;
};
```

No campo pointer temos o valor que é guardado no array, neste caso um comando, existe um counter que contêm o tamanho atual do array e o size com o tamanho efetivo do array.

De maneira a facilitar o trabalho sobre os comandos a executar e seus outputs foi criada uma estrutura auxiliar () onde são guardados os comandos após o parser dos mesmos. Para efetuar o parser foi usada a função strtok e outra funções auxiliares, para remover espaços extra e para tirar o número que vem na execução do comando de maneira a ser possível obter o output das dependências do mesmo , de maneira a remover todas as ocorrências de "$","\n","$",">>>","<<<".

Estrutura esta denominada por **command:**

```
struct command{
    char* doc;
    int dep;
    char* cmd;
    char** pro_cmd;
    char* output;
};
```

Na variável **doc** é guardada a descrição do comando, na **dep** é guardado o número de comandos que é necessário recuar e obter o seu output,para que este possa ser passado como input do comando atual, na variável **pro_cmd** vão ser guardados todos os comandos a serem executados (incluindo os outputs de outros

programas caso a dep seja diferente de 0) e no output é guardado o **output** final do comando em si**.**

Após isto é chamada a **exec_cmd_array** que é a aplicação em si, onde são executadas as funcionalidades do programa.

Dentro desta vai ser percorrido a struct array que é referida em cima,caso existam dependências o comando é executado sem qualquer “input” de outra maneira é passado para uma string de strings os outputs que vão servir como “input” para o comando atual.Após este pequeno processamento da informação é chamada a função **exec_cmd** e é nela que reside o segredo da otimização e das funcionalidades da aplicação.

```
char* exec_cmd(char** cmd, char* input){
    int i, fdf, in[2], out[2];
    char* output = malloc(sizeof(char)*1024);


    pipe(in);
    pipe(out);
    fdf = fork();

    if(!fdf){ // FILHO
        close(in[1]);
        close(out[0]);
        dup2(in[0],0);
        close(in[0]);
        dup2(out[1],1);
        close(out[1]);

        execvp(cmd[0],cmd);

        perror("Error execvp");
        _exit(-1);
    }
    // PAI
    else{

        close(in[0]);
        close(out[1]);
        write(in[1], input, strlen(input));
        close(in[1]);
        wait(&i);
        read(out[0], output, 1024);
    }
    return output;
}
```

Como é possível verificar são usados 2 pipes, um (in) para passar um possível input (vindo de uma possível dependência) para o comando para ser executado e um (out) outro que permite passar o resultado do execvp.Para que isto seja possível são utilizados os redirecionadores de “file descriptor” dup2, de maneira a trocar as extremidades dos pipes que estão a ser usadas pelo “stdin”(0) e “stdout”(1).

Após isto é executado a **print_cmd_file** que nada mais faz que percorrer todo o array e atualizar o ficheiro inicial com o resultado da execução dos programas (atualizando a parte dos outputs).

3 - Funcionalidades

3.1 Execução de programas : A aplicação desenvolvida é capaz de detectar um comando que lhe seja passado no ficheiro, assim como executá-lo e guardar o seu output no ficheiro original.Para tal o comando apenas precisa de estar identificado com o símbolo $.

3.2 Re-processamento de um notebook : A aplicação é capaz de, após um primeiro processamento e execução dos comandos respectivos, executar um novo processamento caso o utilizador altere o ficheiro processado previamente.

3.3 Detecção de erros e interrupção da aplicação : A aplicação é capaz de detetar erros ao longo do processamento e parar a execução por esse motivo, mas a opção de paragem tendo como recurso o ˆC não foi implementada.

3.4 Acesso a resultados de comandos arbitrários : A aplicação tem a capacidade de aceder a outputs de comandos que tenham sido executados anteriormente, graças à maneira como “montamos” a estrutura do trabalho e o campo **dep** e **pro_cmd** falado no ponto 2.O facto de os outputs dos comandos serem todos guardados faz com que não tenha que haver nova execução dos mesmos apenas uma procura pelos seus outputs.Para que tal aconteça é necessário que os comandos sejam precedidos por $n| em que o n é um qualquer número >1 e menor que o número de comandos executados até aquele ponto do ficheiro.

3.5 Execução de conjuntos de comandos : Esta funcionalidade não foi implementada.

4 - Makefile

make : comando que tem como função compilar o programa e gerar os executáveis correspondente ao notebook.

make clean : comando que tem como função remover todos os ficheiros gerados
pela compilação assim como o executável do notebook.

Para a utilização do programa basta usar o comando make e executar o notebook tendo como argumento um ficheiro do tipo **.nb**.

```
IDIR=../include
ODIR=obj
LDIR=lib
OLDIR=$(ODIR)/lib

CC=gcc
CFLAGS = -Wall -std=c11 -std=gnu99 -g -I$(IDIR)

DEPS=$(IDIR)/$(wildcard *.h)
SOURCES=$(wildcard *.c)
MY_LIBS=$(wildcard $(LDIR)/*.c)
SOURCES_OBJ=$(patsubst %.c,$(ODIR)/%.o,$(SOURCES))
MY_LIBS_OBJ=$(foreach o, $(patsubst %.c,%.o,$(MY_LIBS)), $(ODIR)/$o)

print-% : ; @echo $* = $($*)

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

program: $(SOURCES_OBJ) $(MY_LIBS_OBJ)
  $(CC) $(CFLAGS) $(wildcard $(ODIR)/*.o)  $(wildcard $(OLDIR)/*.o) -o notebook

clean:
  rm obj/*.o
  rm obj/lib/*.o
  rm notebook
```

5 - Conclusão

O desenvolvimento deste projeto ajudou a desenvolver os conhecimentos aprendidos durante as aulas de Sistemas Operativos ao longo do semestre, permitiu aplicar um pouco de todos os campos abordados nos guiões e juntá -los numa só aplicação.

As maiores dificuldades estiveram no parser e na sua implementação, optou-se por usar uma estrutura auxiliar e a função strtok para “limpar” e guardar os comandos e respectivos outputs.Funções estas que nos ocuparam grande parte do tempo a resolver problemas de gestão de memória e a tentar perceber de que maneira as poderíamos utilizar.

Apesar de tudo foram implementadas quase todas as funcionalidades,à excepção da execução de conjuntos de comandos.