Universidade do Minho
Escola de Engenharia

Processamento de Notebooks

A75780 - António Jorge Monteiro Chaves
A74745 - Carlos José Gomes Campos
A74166 - Pedro Daniel Gomes Fonseca

Maio 2018

Conteúdo

1 Introdução **4**

2 Funcionalidades básicas **5**
2.1 Execução de programas 5
2.2 Re-processamento de um *notebook* 6
2.3 Deteção de erros e interrupção da execução 7

3 Funcionalidade avançadas **7**
3.1 Acesso a resultados de comandos anteriores arbitrários 7
3.2 Execução de conjuntos de comandos 8

4 Casos de Teste **9**

5 Conclusão **10**

Lista de Figuras

1 Criação dos ficheiros *result.txt* e *errors.txt* 5
2 Deteção dos carateres de delimitação 5
3 Fragmento de código da main onde se cria cada processo para cada *notebook* 6
4 Fragmento de código responsável pela da criação dos vários *pipes* 6
5 Fragmento de código responsável pelo *tokenizer* 6
6 Remoção do ficheiro de entrada pelo *result* 7
7 Remoção dos ficheiros auxiliares 7
8 Deteção do dígito passado como argumento 8
9 Execução após deteção do *input* 8
10 Caso de teste 4 onde é esperado dar erro 9
11 Output no ficheiro *erros.txt* 9
12 Fragmento de código onde deu o erro 9
13 Fragmento de código onde deu o valor esperado 10
14 Fragmento de código onde deu o valor esperado 10

1 Introdução

Como requerido pelo enunciado, iremos proceder de seguida a uma descrição detalhada da forma como realizamos o trabalho, cujo foco principal é o processamento de *notebooks*.

Contextualizando, um *notebook* é um ficheiro de texto que depois de processado é modificado de modo a incorporar resultados da execução de código ou comandos nele embebidos.

As principais funcionalidades implementadas vão de encontro ao que foi pedido no enunciado. Começando pela execução dos comandos que estejam no *notebook*, cujo resultado deverá ser inserido de novo nesse ficheiro. Estes também poderão ser executados com auxilio de comandos anteriores. Implementa ainda a funcionalidade de re-processamento dos *notebooks* devendo fazer a substituição dos novos resultados no ficheiro. Possui ainda a funcionalidade de deteção de erros na execução dos comandos terminando o programa sem alterar o *notebook*, permite ainda alternativamente que o utilizador interrompa o processamento dos comandos, mantendo o *notebook* inalterado.

Numa segunda parte do enunciado, é pedido o que se denominam de funcionalidades avançadas. A principal funcionalidade que implementamos desta fase é o acesso a comandos anteriores arbitrários. Por fim a última funcionalidade pedida no enunciado é a execução de um conjunto de comandos, no entanto não nos foi possível realizar a mesma devido a certos constrangimentos, como dito no enunciado (acréscimo de classificação)/(tempo dispendido) não nos aparentou dar um valor superior a 1 e como tal a decisão do grupo foi unânime e optamos pela não implementação desta funcionalidade.

2 Funcionalidades básicas

2.1 Execução de programas

Quanto à primeira funcionalidade pedida, todas as linhas começadas pelo símbolo $ devem ser interpretadas como comandos (programa e argumentos). Caso a linha comece com $ | , o *input* deve ser o resultado da execução do comando anterior. De seguida o resultado da execução deve ser inserido no ficheiro imediatamente a seguir ao comando respetivo e delimitado por >>> e <<<, sendo que tudo o que estiver entre esses símbolos deverá ser ignorado no que diz respeito a uma nova execução.

O primeiro passo que fazemos para atingir este objetivo é criarmos dois ficheiros, *result.txt* e *errors.txt*, onde no primeiro vamos escrever o resultado da execução dos comandos presentes no ficheiro inicial, bem como os próprios comandos e ainda os comentários. O segundo surge da necessidade da deteção de erros que será explicada mais à frente.

```
void process(char const* name){
    int pp[2];
    char c;
    int i=0,rd=0,wt=0,dupinput=0,duppipe=0,cl=0;
    int in=open(name,O_RDWR,0666),out=open("result.txt",O_CREAT|O_RDWR|O_TRUNC,0666),err=open("errors.txt",O_CREAT|O_RDWR|O_TRUNC,0666);
    int duperr=dup2(err,2);
```

Figura 1: Criação dos ficheiros *result.txt* e *errors.txt*

Por forma a facilitar o tratamento do *input*, sempre que escrevemos o resultado da execução de um comando adicionamos uma \ antes das >>> , e como tal se encontrarmos uma \ ele irá ignorar e não interpretará como um comando pois são considerados como comentários.

```
if(c=='/'){
    wt=write(out,&c,1);
    if(wt<0){
        perror("Erro no write\n");
        remover(i);
        _exit(-7);
    }
    while(rd=read(in,&c,1)){
        if(rd<0){
            perror("Erro no read\n");
            remover(i);
            _exit(-8);
        }
        wt=write(out,&c,1);
        if(wt<0){
            perror("Erro no write\n");
            remover(i);
            _exit(-9);
        }
        if(c=='\n')
            break;
    }
```

Figura 2: Deteção dos carateres de delimitação

No que diz respeito à execução do nosso programa, optamos por não utilizar concorrência paralela, pois tendo em conta que escrevemos todos os erros num único ficheiro pois desta forma garantimos ordem e que o tempo de execução ronda a casa dos milissegundos, não seria muito vantajoso e como tal optamos por uma execução sequencial.

Como tal para o processamento de cada um dos *notebook* criamos um processo, como representado pela figura 3.

```
int f=0;
for(int i=1;i<argc;i++){
    f=fork();
    if(f==0){
        process(argv[i]);
        _exit(0);
    }
```

Figura 3: Fragmento de código da main onde se cria cada processo para cada *notebook*

Aqui, o processo pai fica apenas encarregado pela escrita no ficheiro *result.txt*, que como referido anteriormente tomará o lugar do *notebook* caso tudo corra como planeado, mas também nos ficheiros auxiliares criados para a escrita dos resultados de cada um dos comandos.

Ou seja, os processos filho responsáveis pela execução dos comandos, depois de terminarem enviam para o *pipe* de forma a que o processo pai proceda à escrita dos resultados. De referir que para a execução de cada comando é criado um *pipe*.

```
int ppflag=pipe(pp);
if(ppflag<0){
    perror("Erro no pipe\n");
    remover(i);
    _exit(-15);
}
int f=fork();
```

Figura 4: Fragmento de código responsável pela da criação dos vários *pipes*

Os processos filho como referido anteriormente tratam vários tipos de comandos e atuam mediante o *input* que receberem.

No primeiro caso em que encontra um | , o ficheiro de leitura do processo filho será o i-1.txt, ou seja o resultado da execução do comando anterior, e o restante processo decorre como referido anteriormente com a escrita para o pipe.

No segundo caso que é quando encontramos um dígito no início do comando, será semelhante. No entanto será explicado posteriormente.

Outro aspeto importante referir é o *tokenizer* dos comandos, que posteriormente é utilizado no *execvp*. O que este *tokenizer* faz é basicamente pegar numa string e cria um array de strings delimitados pelo carater "espaço".

```
for(token=strtok(tmp," ");token;k++,token=strtok(NULL," ")){
    cmd=(char**)realloc(cmd,(k+1)*sizeof(char*));
    cmd[k]=strdup(token);
}
```

Figura 5: Fragmento de código responsável pelo *tokenizer*

2.2 Re-processamento de um *notebook*

Quanto ao re-processamento de um *notebook*, como referido anteriormente com a adição da \ antes das <<<, quando o utilizador quiser modificar algum comando e executar novamente, ele quando encontrar o carater \ irá ignorar tudo o que estiver delimitado por <<< e >>>.

Esta funcionalidade é garantida graças ao ficheiro *result.txt* onde escrevemos todos os resultados da execução dos variados comandos e como tal conseguimos garantir o re-processamento de um ou mais *notebooks*.

2.3 Deteção de erros e interrupção da execução

No que toca à deteção de erros, como referido anteriormente nós criámos um ficheiro chamado *errors.txt* que caso algum dos *perror* que temos ao longo do código faça print de uma mensagem de erro para o *stderr*, o programa aborta e, é escrito nesse ficheiro tudo o que foi mandado para o *stderr* e ainda, como implementamos que a execução dos comandos será reescrita no *result.txt* e apenas no final se fará a substituição do ficheiro de entrada pelo auxiliar, o *notebook* permanecerá inalterado.

Outra necessidade exposta no enunciado diz respeito à possibilidade do utilizador interromper um processamento em curso.

O método implementado para a deteção de erros, sempre que for detetada uma *system call* que foi invocada pelo processo pai e deu erro, faz-se uma chamada à função remover, que remove todos os ficheiros auxiliares até ao momento criados. No caso de ter ocorrido no processo filho o procedimento é o mesmo e faz-se ainda *kill* ao processo pai.

```
    remove(name);
    rename("result.txt",name);
```

Figura 6: Remoção do ficheiro de entrada pelo *result*

```
void remover(int i){
    for(int z=0;z<=i;z++){
        char* n4=NULL;
        asprintf(&n4,"%d.txt",z);
        remove(n4);
    }
    remove("result.txt");
}
```

Figura 7: Remoção dos ficheiros auxiliares

3 Funcionalidade avançadas

3.1 Acesso a resultados de comandos anteriores arbitrários

No que diz respeito à primeira e única funcionalidade avançada que implementamos no nosso código, acesso a resultados anteriores arbitrários, linhas que começem por $n| ao invés de ter como *input* os resultados da execução do comando anterior, devem ter o resultado da execução do n-ésimo comando anterior.

Ou seja, neste caso o ficheiro de leitura será o i-n.txt e o processamento será igual aos restantes, depois da execução do comando, o processo filho envia para o pipe os resultados para que o processo pai faça a escrita dos mesmos.

De seguida apresentamos um fragmento de código onde verificamos se é um dígito o *input* recebido e posteriormente o fragmento de código onde executamos o comando.

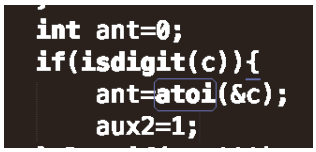

Figura 8: Deteção do dígito passado como argumento

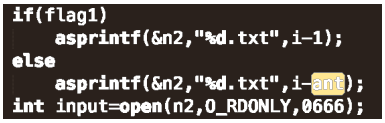

Figura 9: Execução após deteção do *input*

3.2 Execução de conjuntos de comandos

Como referido na introdução do trabalho, considerando a equação (acréscimo de classificação)/(tempo dispendido), optamos por não proceder à implementação desta funcionalidade pois de facto tendo em conta o *deadline* da entrega do trabalho não nos seria possível implementar esta funcionalidade da forma pretendida pelos docentes.

4 Casos de Teste

Como requerido pela equipa docente, criamos um conjunto de casos de teste de forma a demonstrar os resultados esperados para cada input recebido. Criámos casos em que o resultado é positivo e outros em que é esperado dar erros.

```
/ Este comando lista os ficheiros:
$ ls
/ Vai dar erro, so não é um comando:
$1 | so
/ E escolher o primeiro:
$2 | head -1
```

Figura 10: Caso de teste 4 onde é esperado dar erro

Como podemos observar, o comando "so"não existe e como tal, é esperado que a execução dê erro.

```
File: exemplo4.nb
Erro no exec
: No such file or directory
```

Figura 11: Output no ficheiro *erros.txt*

```
execvp(cmd[0],cmd);
handler(i);
perror("Erro no exec\n");
_exit(-23);
```

Figura 12: Fragmento de código onde deu o erro

Apresentamos agora um output que seria o esperado.

```
/ Este comando lista os ficheiros:
$ ls
>>>
enunciado-so-2017-18.pdf
errors.txt
exemplo1.nb
exemplo2.nb
exemplo3.nb
exemplo4.nb
exemplo5.nb
exemplo6.nb
makefile
processNotebooks
processNotebooks.c
result.txt
<<<
/ Este comando mostra o conteudo da makefile
$ cat makefile
>>>
processNotebooks:   processNotebooks.c
                    gcc -o processNotebooks processNotebooks.c

execute:            processNotebooks
                    ./processNotebooks *.nb

install:            processNotebooks
                    sudo cp processNotebooks /usr/local/bin

remove:             processNotebooks
                    rm processNotebooks errors.txt<<<
/processNotebooks *.nb
/usr/local/bin
/processNotebooks *.nb
/usr/local/bin
/processNotebooks *.nb
/usr/local/bin
/processNotebooks *.nb
/usr/local/bin
```

Figura 13: Fragmento de código onde deu o valor esperado

```
/ Este comando conta o numero de palavras do enunciado
$ wc -c enunciado-so-2017-18.pdf
>>>
74687 enunciado-so-2017-18.pdf
<<<
```

Figura 14: Fragmento de código onde deu o valor esperado

5 Conclusão

Fazendo uma retrospetiva ao nosso trabalho realizado durante toda a execução deste trabalho prático, denotamos alguns aspetos a melhorar.

Primeiramente, poderíamos ter implementado uma execução concorrente, no entanto não o fizemos pois apenas utilizamos um ficheiro de erros e como tal a execução teria de ser sequencial de forma a que fosse percetível onde teriam ocorridos os erros.

Sublinhamos também, que se calhar o método implementado em que pomos um processo filho a matar um processo pai não é o mais convencional, mas que consideramos um método simples e eficaz.

Em suma, apenas não realizamos uma das alíneas propostas e reconhecemos que existem alguns aspetos que poderiam ter outra implementação mas que devido a vários fatores escolhemos não o fazer.