Trabalho de Sistemas Operativos
Processamento de um notebook

Sérgio Oliveira
a62134

Pedro Dias
a63389

21 de Maio de 2018

Contents

Chapter 1

Introdução

Chapter 2

Execucão

2.1 Noções básicas

A funcionalidade básica do nosso programa está em ler o nosso *notebook* linha a linha e termina a sua execução até que o ficheiro passado como argumento tenha sido lido pelo *while* até à linha final.

Como podem existem caractéres especiais no início de cada linha de código de um ficheiro, primeiro temos que fazer o processamento da mesma, filtrando os conteúdos para as variáveis corretas.

Imaginemos este exemplo como *stdin* retirado do enunciado:

```
Este comando lista os ficheiros:
$ ls
```

O programa irá fazer o *parsing* das linhas e estas são processadas para que se reconheça o conjunto de caratéres especiais para o correto processamento do ficheiro, sendo eles:

- $;
- $|;
- $(número)|;
- >>> e <<<.

Uma linha delimitada por qualquer outra expressão diferente dos itens em cima irá ser ignorada e não interpretada como comando (mantida intacta no ficheiro original).

Para cada caratér especial, temos então variáveis guardadas para podermos fazer comparações com o ficheiro *notebook*. Neste caso, consideremos a variável '$' (char * dollar).

```
if(strncmp(dollar,line,strlen(dollar))==0 && (running)){
```

Através do *if* acima referido e:

```
char * trim(char * s) {
    int l = strlen(s);
    while(isspace(s[l - 1])) --l;
    while(* s && isspace(* s)) ++s, --l;
    return strndup(s, l);
}
```

através da função trim, podemos então ignorar o que não é necessário na nossa linha e executamos o comando 'ls' através da função 'execl'.

2.2 Ciclo principal

O cíclo while que está representado na função 'main' é o motor de todo o nosso programa, que está a ser controlado através de flags e da função read, que ao ser executada, retorna o valor de bytes que foram lidos. Se este valor retornado for negativo, então o *system call* está a retornar um erro.

Esta é então a nossa maneira de parar o cíclo, executando-o até que o nosso read fique sem mais bytes para ler.

```
    while ((read = getline(&line, &len, fp)) != -1 && flagErrorFork
       ==1 && (running)) {
```

Portanto, se este cíclo terminar com as respectivas flags a retornarem sucesso na saída, o programa substitui o ficheiro processado pelo programa pelo ficheiro *notebook* original através da função 'rename' e remove então a pasta temporária já não necessária para o processamento (iremos falar mais tarde sobre a sua utilidade).

Se o cíclo não retornar sucesso, a função 'rename' já não será executada para que o ficheiro *notebook* original não seja alterado e remove tudo o que já não é necessário.

2.3 Re-processamento

A funcionalidade normal do nosso programa será sempre executar e inserir os nossos resultados entre >>> e <<<. Haverá no entanto alturas em que faremos alterações ao nosso sistema (criamos um ficheiro novo, o *word count* de determinado ficheiro é agora maior, o estado de X dispositivo foi alterado, entre outros).

Nestes casos, necessitamos então de voltar a executar as linhas de comando do nosso *notebook*.

```
void re_processamento(char * file){
```

A função re-processamento abre então um ficheiro temporário para podermos fazer o *parsing* do nosso ficheiro de entrada, fazendo com que o original não seja imediatamente alterado.

```
    FILE *REDO;
    REDO = fopen("REDO.txt", "wr+");
```

Executamos então outro ciclo da mesma natureza do *while* da função main, lendo linha a linha e, desta vez, ignorando todo o conteudo entre >>> e <<< do ficheiro de entrada.

Após isto tudo, usámos a função rename para relocarmos o ficheiro temporário para o original.

```
    rename("REDO.txt",file);
```

2.4 Detecção de erros / Interrupção

Uma das flags é usada para depuração de erros de qualquer fork criado. Se o fork retorna um *status* diferente de sucesso, pode significar que a linha lida contém um comando errado. Neste caso, teriamos então de cancelar a execução do nosso *notebook*.

Para resolver isso, recorremos à função forkError,

```
int forkError(int status,char *b){
```

que, ao longo de toda a nossa função main, determina o valor da variável local flagForkError.

A função forkError faz o teste, através de estados de finalização de processos, para que possámos apanhar o estado correto de cada fork criado e sabermos então se podemos interromper o programa ou não.

No entanto também pode haver ação humana e, para isso, o sinal *SIGINT* é enviado para o programa (sinal normalmente relacionado com a combinação de botões *Control+C*). Nesse caso, a nossa variável global *running* irá determinar o estado de execução do nosso programa.

```
static volatile int running = 1;
void handler(int dummy){
    running = 0;
}
```

A detecção do sinal deve ser inicializada na main pela função de sistema *signal*.

Chapter 3

Outras Funcionalidades

3.1 Histórico de comandos anteriores

Nesta secção, decidimos usar a biblioteca Regex.

```
#include <regex.h>
```

A funcionalidade das expressões regulares para podermos comparar com o conjunto de caratéres especiais `$n|` era o ideal para podermos continuar a desenvolver esta parte do trabalho.

A partir daqui, houve duas ideias diferentes para o que haveriamos de fazer. Uma das ideias seria a criação de um ficheiro num formato básico de texto chamado *history*, em que ao longo da leitura das linhas com comandos do nosso *notebook*, o programa copiava o comando e colava como uma nova linha no nosso histórico de comanados.

Quando o programa reconhecesse o conjunto de caratéres especiais `$n|`, guardávamos 'n' numa variável como o número do comando da lista de comandos anteriores no *notebook*. 'n' seria então o número da linha existente no nosso ficheiro *history* para quando quiséssemos que fosse executada.

Em vez desse procedimento, optamos por outra solução.

Dada uma linha de comando executada pelo programa, é então criado um ficheiro (de seu nome resultN.txt) para que o output desse mesmo comando fosse armazenado.

a variável N no nome do ficheiro é uma variável (int), significando então o número da linha que foi executada.

Através de funções da biblioteca *string.h*, conseguimos converter a variável N para que possámos escrever e ler corretamente os ficheiros resultN.

```
void strcatFilename(int contador){
    char append[50];
    char contadorString[5];
    strcpy (filename, "tmp/result");
    strcpy (append, ".txt");
    sprintf(contadorString,"%d",contador); // int to string
    strcat(filename, contadorString);
    strcat(filename,append);
}
```

```
strcatFilename(contador); // ABRIR RESULTN.TXT
```

Em resumo, temos então a mesma quantidade de linhas de comando executadas e ficheiros resultN, para o output ser usado de várias maneiras possíveis, como veremos na secção seguinte.

3.2 Execução de conjuntos de comandos

Resta-nos então falar de um conjunto de caratéres especiais referido anteriormente. A expressão $| remete-nos para uma execução em pipe e está guardada numa variável local no nosso programa (char * dollarPipe) para que possámos mais uma vez fazer as comparações corretamente.

```
        if(strncmp(dollarPipe,line,strlen(dollarPipe))==0 && (running
            )){
```

Mais uma vez através da função trim, "cortamos" o que não é necessário na nossa linha para executar o comando.

```
char * trim(char * s) {
    int l = strlen(s);
    while(isspace(s[l - 1])) --l;
    while(* s && isspace(* s)) ++s, --l;
    return strndup(s, l);
}
```

No entanto, desta vez, precisamos do comando anterior a este, visto que é uma execução em pipeline.

Usando o método da secção anterior, como estamos então a guardar o output dos comandos que executamos anteriormente, conseguimos ler o output do comando anterior.

Para isso decrementamos a variável N para podermos aceder ao ficheiro resultN correto do comando anterior.

```
                    strcatFilename(contador-1);
```

A partir daí, com a system call fork(), o processo pai e o processo filho fazem então a execução em pipeline.

Chapter 4

Conclusão

Após várias semanas de desenvolvimento, há que realçar alguns pontos importantes, como a certa dificuldade em utilizar funções de baixo nível (implementação de pipes com nome, correta comunicação entre forks, entre outros).

No entanto, houve sempre boas ideias vários e boa comunicação, e após correcções e vários testes ao nosso trabalho, estamos satisfeitos por ter ultrapassado esses entraves e ter concluído o projeto.

Ficamos a compreender mais sobre as system calls de baixo nível e em que contexto usá-las, para termos um maior controlo de processos e ficheiros.

Com o desenvolvimento destes projetos, haverá também sempre a expansão do nosso conhecimento e interesse em sistemas UNIX e UNIX-like.