Trabalho Prático de
Sistemas Operativos

Processador de Notebooks

2017/2018

- GRUPO 42 -
Vitor Peixoto (a79175)
Francisco Oliveira (a78416)
Raul Vilas Boas (a79617)

Braga, 2 de Junho de 2018

1 Introdução

Como trabalho prático da unidade curricular de Sistemas Operativos, foi-nos pedido para construir um sistema de processamento de *notebooks*.

Estes *notebooks* misturam documentação, linhas de código e resultados de execuções anteriores. Neste contexto, um *notebook* é um ficheiro de texto que, depois de processado, é modificado de modo a incorporar resultados da execução de código.

No nosso caso, optamos por adotar uma estratégia que se baseia numa implementação para sistemas *Unix*, onde linhas começadas por `"$"` seriam linhas com comandos a executar, zonas delimitadas por `">>>"` antes e `"<<<"` após seriam outputs de execuções anteriores, e qualquer outro texto seriam comentários e/ou anotações do documento.

2 Desenvolvimento da Solução

Assim, no âmbito deste sistema de processamento de *notebooks*, foi-nos proposto que este fosse desenvolvido tendo em conta as seguintes três funcionalidades principais:

- Executar os comandos;
- Re-processar antigos *notebooks*;
- Não atualizar *notebooks* em caso de falha ou interrupção de execução.

2.1 Objetivo 1 - Execução

O primeiro passo para o desenvolvimento do sistema passa por desenvolver um processador de *notebooks*.

Para facilitar a execução do sistema, foi desenvolvida uma *Makefile*:

```
teste : notebook
        ./notebook teste.nb

notebook : code.c
        gcc -o notebook code.c
```

Esta *Makefile* compila o ficheiro de código C e executa-o com o ficheiro de teste como argumento.

Relativamente ao processamento do *notebook*, o ficheiro de teste - passado como argumento - é aberto e começa a ler um carater de cada vez. Sempre que encontrar `"$"` irá tratar o resto da linha como um comando a executar.

Fará o mesmo quando encontrar `"$|"` contudo irá receber como input no *stdin* o output do último comando executado.

Para executar então a linha, primeiramente criamos dois pipes (*'inputFilho'* e *'filhoPai'*) e depois usamos o comando *fork* para criamos um filho.

Os pipes irão permitir a troca de informação entre o pai e o filho. O *'inputFilho'* será responsável por passar dados do pai para o filho e o *'filhoPai'* será responsável por passar do filho para o pai.

O filho vai então depois correr o comando que estava na linha. O output gerado será redirecionado usando o pipe *'filhoPai'* para o pai.

Caso seja uma linha de `"$|"` o filho vai receber o último output no *stdin* através do pipe *'inputFilho'*.

O pai entretanto espera que o filho 'morra' e vai depois ler o output no pipe *'filhoPai'*, guardando-o no buffer (estrutura auxiliar responsável por guardar toda a informação necessária para atualização do *notebook* original).

Caso seja uma linha de `"$|"` o pai vai também enviar o último output para o filho receber como *stdin* pelo pipe *'inputFilho'* antes de esperar que ele 'morra'.

Na figura abaixo conseguimos ver o esquema de interação entre o pai e o filho através dos pipes:

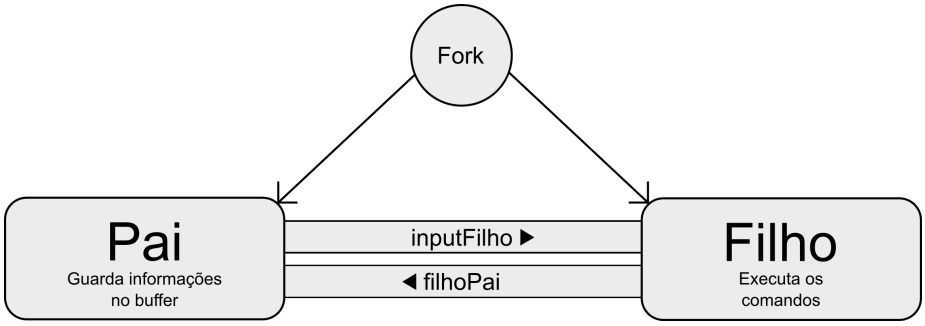

Figura 1: Esquema da interação Pai-Filho com *pipes*

Abaixo temos as partes mais importantes do código que foram explicadas acima, sendo que o código está devidamente documentado para fácil entendimento e ligação lógica com a explicação:

```
if( (ant1 == '$' && c == ' ') ||
    (ant2 == '$' && ant1 == '|' && c == ' ') ){
        (...)
        // criação de pipes
    int inputFilho[2];
    pipe(inputFilho)
    int filhoPai[2];
    pipe(filhoPai);
    int childpid = fork(); // fork

    if(childpid==0){ //child
        // antes do exec alteramos o stdin e stdout
        if( ant2 == '$' && ant1 == '|' && c == ' ' ){
            close(inputFilho[1]);
            dup2(inputFilho[0], STDIN_FILENO);
            close(inputFilho[0]);
        }
        close(filhoPai[0]);
        dup2(filhoPai[1], STDOUT_FILENO);
        close(filhoPai[1]);
        executa(str); // executa o comando
        exit(-1);
    }
    //CONTINUA NA PRÓXIMA PÁGINA
```

```
    //CONTINUAÇÃO

    else{ //parent
        // aqui enviamos input para o filho
        if( ant2 == '$' && ant1 == '|' && c == ' ' ){
            close(inputFilho[0]);
            write(inputFilho[1],
                buffer+lastOutStart, lastOutSize);
            close(inputFilho[1]);
        }

        int filho = wait(&status);
        if(status) // status!=0 problemas no último filho
            exit(-1);

        // zona de leitura do output do filho
        buffer[i++]='>';buffer[i++]='>';
        buffer[i++]='>';buffer[i++]='\n';
        lastOutStart=i;
        close(filhoPai[1]);
        for(int nread=0; (nread=read(filhoPai[0], pipebuffer,
            sizeof(pipebuffer))) != 0;)
            for(j=0; j<nread; j++)
                buffer[i++]=pipebuffer[j];

        close(filhoPai[0]);
        lastOutEnd = i;
        lastOutSize = lastOutEnd-lastOutStart;
        buffer[i++]='<';buffer[i++]='<';
        buffer[i++]='<';buffer[i++]='\n';
    }
}
```

A seguir temos uma demonstração do funcionamento do processador de *notebooks* quando aplicado ao ficheiro de teste.

A primeira imagem mostra o *notebook* antes de ser executado, sem qualquer output nele, e a segunda mostra o mesmo depois de executado, com os outputs dos respetivos comandos.

Estes outputs foram os devolvidos pelo filho durante a execução, recebidos pelo pai, guardados no buffer e finalmente no fim da execução o *notebook* é atualizado com os outputs, imediatamente após ao comando que lhes deu origem.

Ficheiro *teste.nb* antes de executar:

```
Este comando lista os ficheiros na pasta:
$ ls
Ordenamos por ordem alfabética decrescente:
$| sort -r
```

Ficheiro *teste.nb* após executar:

```
Este comando lista os ficheiros na pasta:
$ ls
>>>
Relatorio
code.c
enunciado-so-2017-18.pdf
makefile
notebook
teste.nb
<<<
Ordenamos por ordem alfabética decrescente:
$| sort -r
>>>
teste.nb
notebook
makefile
enunciado-so-2017-18.pdf
code.c
Relatorio
<<<
```

2.2 Objetivo 2 - Re-Processamento

Como segundo ponto deste trabalho prático, era-nos pedido que, se voltássemos a correr o *notebook*, com os mesmo comandos ou até alguns deles alterados, ele ignorasse os antigos outputs resultantes da última execução e colocasse no seu lugar o novo output resultante desta nova execução.

Este processo é bastante simples. Para tal, durante a execução, não guardávamos o antigo output no buffer mas sim o novo output resultante da execução devolvida pelo filho.

Como podemos ver no excerto de código abaixo, caso encontremos a marca de inicialização de comentário, colocamos *comment*=1 (isto significa que estamos dentro de um output). Assim, a partir do momento em que entramos num output, ignorámos todo o texto do ficheiro até encontrarmos a marca de finalização de output. Assim que o encontramos colocamos o *comment*=0 e continuamos a leitura do ficheiro normalmente.

Código referente ao tratamento de outputs:

```
if(comment){
        if( ant2=='<' && ant1=='<' && c=='<' ){
                comment=0;
                fgetc(file); // retira o \n após o <<<
        }
}
else{
        buffer[i++] = c; // guarda o carater atual
        if( ant2=='>' && ant1=='>' && c=='>' ){
                comment=1;
                buffer[i-1]='\0';
                buffer[i-2]='\0';
                buffer[i-3]='\0';
                i-=3;
        }
        (...)
```

Podemos aqui também ver como o re-processamento ignora o antigo output e o substitui pelo novo.

Ficheiro *teste.nb* antes de executar:

```
Este comando lista os ficheiros na pasta:
$ ls
>>>
Relatorio
code.c
enunciado-so-2017-18.pdf
makefile
notebook
teste.nb
<<<
Aqui era um 'sort -r' e trocamos por um 'wordcount':
$| wc
>>>
teste.nb
notebook
makefile
enunciado-so-2017-18.pdf
code.c
Relatorio
<<<
```

Ficheiro *teste.nb* após executar:

```
Este comando lista os ficheiros na pasta:
$ ls
>>>
Relatorio
code.c
enunciado-so-2017-18.pdf
makefile
notebook
teste.nb
<<<
Aqui era um 'sort -r' e trocamos por um 'wordcount':
$| wc
>>>
       6       6      69
<<<
```

2.3 Objetivo 3 - Erros e Interrupção

Como último ponto, era pedido que, quando a execução fosse interrompida, devido a erro no *exec* ou insucesso de execução do programa, o processamento deveria ser interrompido e o *notebook* deveria permanecer inalterado.

Para tal, caso o filho não conseguisse executar o comando ou falhasse por qualquer motivo este iria enviar uma mensagem de erro e terminar. Por consequência, o pai - que estava à espera que o filho morresse - verifica o valor do *status* devolvido pela função *wait* e caso este seja diferente de 0, sabe que o filho não foi sucedido na execução e então dá *exit(-1)*, terminando a execução e notificando o erro.

Abaixo temos um excerto da função *executa*:

```
int executa(char* cmd){
    (...)
    execvp(args[0],args);

    //se o exec falhar continua e devolve o erro.
    perror(">ERROR: exec() ");
    return -1;
}
```

Outro caso onde a execução pode ser é interrompida é caso seja inserido um Ctrl+C a meio da execução. Para a interromper a execução usando o Ctrl+C, apenas utilizamos um *signal* para este comando em específico, sendo que o sinal de interrupção do Ctrl+C é o `'SIGINT'`. Este sinal é iniciado imediatamente na *main* e caso seja 'acionado' executa a função *terminate* que imprime o erro e interrompe a execução:

```
void terminate(){
        printf("Execução cancelada com Ctrl+C\n");
        _exit(0);
}

int main(int argc, char** argv){
        signal(SIGINT, terminate);
        (...)
```

Podemos ver também nas imagens seguintes um exemplo de um ficheiro de teste com um comando inválido e o respetivo output de erro, por falha no *exec*.

Ficheiro *teste.nb* antes de executar onde temos um comando inválido:

```
Este comando é inválido:
$ istonaoeumcomandovalido
```

Output do terminal aquando da execução:

```
$ make teste
./notebook teste.nb
>ERROR: exec() : No such file or directory
make: *** [teste] Error 255
$
```

Tal como seria de esperar, o ficheiro *teste.nb* permanece inalterado.

3 Conclusão

O sistema de processamento de *notebooks* permite a execução de comandos, armazenamento de outputs e o registo de comentários ou notas relativas aos comandos executados. Isto pode ser bastante útil como método de registo de comandos executados, servindo por exemplo, como documentação de um programa.

Este trabalho prático contribuiu bastante para a consolidação da aprendizagem obtida nas aulas práticas da unidade curricular e também como aquisição de conhecimentos e ferramentas para o futuro.

O resultado final é positivo, pois para além de nos ter permitido desenvolver conhecimentos acerca da unidade curricular, conseguimos também desenvolver um processador de *notebooks* funcional, que cumpre com os objetivos pedidos no enunciado do trabalho prático.