

# Trabalho de Sistemas Operativos

## Processamento de um notebook

Sérgio Oliveira  
a62134

Pedro Dias  
a63389

21 de Maio de 2018

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Execução</b>	<b>3</b>
2.1	Noções básicas . . . . .	3
2.2	Ciclo principal . . . . .	4
2.3	Re-processamento . . . . .	4
2.4	Detecção de erros / Interrupção . . . . .	5
<b>3</b>	<b>Outras Funcionalidades</b>	<b>6</b>
3.1	Histórico de comandos anteriores . . . . .	6
3.2	Execução de conjuntos de comandos . . . . .	6
<b>4</b>	<b>Conclusão</b>	<b>7</b>

# Capítulo 1

## Introdução

Com este relatório iremos demonstrar o desenvolvimento de um programa que faz o processamento de ficheiros no formato designado pelo trabalho prático.

Um ficheiro de formato *notebook* tem a extensão '.nb'. Este pode conter linhas de código para ser interpretadas pela *shell de sistema*, resultados de execução das respetivas linhas de código e outro tipo de conteúdo tal como documentação.

Ao ser executado, o argumento terá de ter um caminho válido para um ficheiro *notebook* para funcionar corretamente.

Relacionado aos temas que falamos, apresentaremos fragmentos de código ao longo do relatório, para que haja melhor compreensão do que estamos a explicar.

Este relatório está dividido em várias secções que correspondem aos pontos mais importantes do trabalho prático, como vemos na página de conteúdos.

## Capítulo 2

# Execução

### 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 existir caracteres 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 caracteres 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 caractere especial, temos então variáveis guardadas para podermos fazer comparações com o ficheiro *notebook*. Neste caso, consideremos a variável '\$' (char \* dollar).

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

Através do *if* acima referido e através da função trim:

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

executamos o comando 'ls' através da função 'execl'.

## 2.2 Ciclo principal

O ciclo `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 ciclo, 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 ciclo 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 ciclo 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", "w+");
```

Executamos então outro ciclo da mesma natureza do *while* da função `main`, lendo linha a linha e, desta vez, ignorando todo o conteúdo 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, teríamos então de cancelar a execução do nosso *notebook*.

Para resolver isso, recorreremos 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 possamos 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*.

## Capítulo 3

# Outras Funcionalidades

3.1 Histórico de comandos anteriores

3.2 Execução de conjuntos de comandos

Capítulo 4

Conclusão