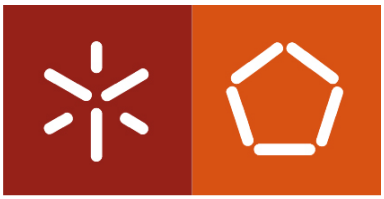

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Sistemas Operativos

Processamento de Notebooks

Trabalho Prático de Sistemas Operativos

Alfredo Gomes A71655
Francisco Costa A70922
João Vieira A71489

2 de Junho de 2018

Conteúdo

Capítulo 1

Introdução

No âmbito da unidade curricular de Sistemas Operativos, foi proposta a realização de um sistema "processador de notebooks"que executa determinados comandos. Este deverá ser capaz de interpretar um ficheiro de texto, executar os comandos que se encontram devidamente formatados nesse ficheiro e guardar o resultado no mesmo ficheiro após o comando, separado por >>> e <<<.

Atendendo a que o "mundo possa vir a ser imperfeito", o sistema deverá ser capaz de parar a execução de comandos e manter o estado inicial do ficheiro caso aconteça um dos dois cenários:

- O comando a executar é impossível, por exemplo, `$cd pastaQueNaoExiste` ou `$comandoQueNaoExiste`
- A execução é travada pelo utilizador, por exemplo, através de um **Ctrl-C**

Capítulo 2

Objetivos do trabalho

2.1 Funcionalidades básicas

Como já brevemente mencionado anteriormente, o sistema deverá ser capaz de identificar e executar comandos que estão presentes num notebook.

No entanto, existem algumas nuances que deverão ter tidas em conta. Os comandos a executar estão sinalizados com um `$` ou `$1` sendo os resultados de execuções previas (linhas entre ¿¿¿ e ¡¡¡) ignoradas e tudo o resto passado diretamente para o novo resultado.

Apesar disso, é possível encontrar comandos devidamente sinalizados dentro do resultado da execução de um ficheiro, por exemplo, quando nos encontramos a processar um notebook que já foi anteriormente processado. Tais comandos serão descartados, uma vez que estes encontram-se dentro de uma região restrita e, como tal, não deve ser tida em conta, quando se procuram comandos para executar.

2.2 Breve explicação da estrutura do trabalho

Quando o Notebook é executado, faz uma leitura do ficheiro linha a linha e cria um ficheiro temporário, a partir de agora designado NBtemp. Caso a frase seja iniciada com um **$** verifica-se se o carácter seguinte é um |. Se não for, o comando no resto da linha é executado e é criado mais um ficheiro temporário, designado por EXtemp, onde serão guardados os resultados desta função em particular, e escreve os resultados para o NBtemp. Se a frase for iniciada por |, vai procurar o último EXtemp criado com ajuda de uma variável global que conta o número de EXtemp, e junta o nome desse ficheiro ao comando a executar, procedendo a executar esse comando e guardar os resultados do modo visto anteriormente. Após a leitura, caso não tenha acontecido qualquer erro e a execução não tenha sido interrompida, o ficheiro original será apagado e substituído pelo NBtemp, caso contrário o NBtemp será apagado e a execução interrompida mantendo-se o ficheiro original inalterado.

2.3 Makefile

Uma **Makefile** é um ficheiro quem contém um conjunto de diretivas usadas pela ferramenta de automação de compilação *make* para gerar um alvo/meta. Esta serve, em alguns casos, para facilitar a interação do utilizador com um programa.

Para este programa, a Makefile será capaz de:

- **make install**: Instalar o programa, onde este é compilado e é aberto um notebook base, no qual o utilizador inserirá os comandos que deseja;
- **make run**: Executar o programa, onde após este ser executado, o notebook base é apresentado ao utilizador para que este possa visualizar o resultado obtido;
- **make reset**: Reiniciar o notebook - Quando se efetua a instalação do programa, após o utilizador inserir os comandos, é criado um ficheiro de *backup* para que seja possível ao utilizador voltar ao estado inicial do notebook, algo conseguido através desta função.
- **make edit**: Editar o notebook, onde o utilizador terá a liberdade de alterar o notebook;
- **make view**: Visualizar o notebook, onde à semelhança do último passo da execução do programa, o utilizador poderá visualizar o conteúdo do notebook.

Capítulo 3

Processador de notebooks

Neste capítulo é feita uma breve explicação da implementação do processador de notebooks.

Resumidamente, o processador de notebooks é um comando que dado um nome de um ficheiro, neste caso, notebook.nb, interpreta-o, executa os comandos nele embebidos, e acrescenta o resultado da execução dos comandos ao ficheiro.

De seguida é apresentado com algum detalhe excertos do código desenvolvido (alguns com comentários que servem de explicação) para que cada funcionalidade fosse atingida com sucesso.

3.1 Execução de programas

Leitura do ficheiro

O primeiro passo deste programa consiste na leitura do ficheiro que se pretende processar.

Assim sendo, usando o operador ternário apresentado de seguida, o ficheiro é aberto caso o número de argumentos que se passa à *main* deste programa (*argc*) for igual a dois (Por exemplo: ./processa notebook.nb). Caso contrário, atribui 0 a *fe* e não executa.

A abertura do ficheiro é feita com a chamada ao sistema da função **open()** sobre o ficheiro que está em *argv[1]*, cujo o resultado será guardado em *fe*. Este resultado diz respeito ao descritor do ficheiro que fica associado que será 3, uma vez que os outros descritores já estão associados (0 standard input, 1 output e 2 standard error).

```
int fe = argc == 2 ? open(argv[1], O_RDONLY) : 0 ;
```

Posteriormente, atribui-se a *new* o resultado do **readAndWriteLine**, que será numa fase posterior o novo ficheiro já processado.

```
char* new = readAndWriteLine(fe);
```

Esta função **char* readAndWriteLine(int fe)** recebe o descritor de ficheiro do notebook que se quer processar.

No seguinte excerto da função *readAndWriteLine* verifica-se que é criado um buffer, que é limpo no início e no fim de cada iteração do ciclo, de maneira a manter a coesão do mesmo. O **sprintf()** é usado para colocar no início do buf um espaço.

A leitura é possível através da implementação de uma função com base na função do exercício 5 do primeiro guião **readln** que recebe o descritor do ficheiro a ler, o apontador para onde está a linha a ser lida e o comprimento (*size of*) do buf-n.

Esta função retorna um *ssize_t* com o número de caracteres lidos. Caso o resultado retornado for 0 significa que o ficheiro foi lido até ao fim (não tem mais nenhuma linha para ler depois desta) e interrompe o ciclo.

```
char buf[1024] ;
int i = 0;

for(i=0; i!=sizeof(buf); buf[i++]='\0');
while(1){
        size_t n = sprintf(buf, "␣");
        ssize_t nb = readln(fe, buf+n, sizeof(buf)-n);


        if(nb==0) break;
        for(i=0; i!=sizeof(buf); buf[i++]='\0');
        }

return file;
```

A função *readln*, cujo protótipo é compativel com a chamada ao sistema **read**, lê de um identificador de ficheiro (*fildes*) e o que lê coloca num sitio na memória (apontador para caracteres). Pode ler no máximo nbytes.

De salientar que são efectuadas chamadas ao sistema com o *read* que lê o caracter em questão e retorna maior que zero se a leitura for efetuada com sucesso e zero caso contrário. Esta é feita para garantir a boa leitura do ficheiro.

```
ssize_t readln(int fildes, void *buf, size_t nbyte){
    ssize_t nb = 0;
    char* p = buf; // aponta para o primeiro char do buffer
    while (1){
        // se o numero de caracteres lidos igualar o maximo de caracteres
        // que se pode ler, interrompe o ciclo
        if(nb==nbyte) break;
        ssize_t n = read(fildes, p ,1);

        // se o resultado do read for 0, interrompe o ciclo
        if(n<=0)break;

        //incrementa o numero de caracteres lidos
        nb+=1;

        // se o caracter for \n e porque chegou ao fim da linha
        if(*p == '\n')break;

        // vai a proxima posicao do buffer p (ou seja, proximo caracter a ler)
        p+=1;
```

```
    }

    return nb;
}
```

Execução de comandos e Escrita no ficheiro

O próximo passo consistiu na implementação da execução de comandos e posterior escrita no ficheiro. Estes são explicados em simultâneo pois foram implementados na mesma altura e complementam-se.

Para perceber o que era necessário fazer começou-se por uma análise e pensamento daquilo que se ia implementar para atingir os objetivos propostos. Chegou-se às seguintes conclusões:

As linhas começadas por $ devem ser interpretadas como comandos (programa e argumentos).

O resultado produzido deve ser inserido no ficheiro, imediatamente a seguir ao comando respetivo, delimitado por >>> e <<<.

Para se atingir estes objetivos, foi acrescentado à função **readWriteLine** o código mostrado de seguida.

Como explicado anteriormente, à medida que se vai lendo as linhas vai-se preenchendo um buffer **buf** com a linha em questão.

É criado um ficheiro temporário para onde se vai efetuar toda a escrita e para não haver conflitos (será explicado na secção de reprocessamento).

Outra consideração importante tem haver com o facto de o utilizador se esquecer de colocar um enter no final da linha, o que pode causar problemas pois se esta for a última linha e tiver um comando, o output será escrito a seguir ao comando na mesma linha e não na seguinte linha, como era pedido e que posteriormente podia ser maléfico para o reprocessamento do ficheiro. Este pequeno problema é ultrapassado através de uma verificação se a linha em causa termina com uma nova linha. Caso não seja coloca-se o caracter em questão.

```
    char* file = "file.txt";

    //chamada ao sistema da funcao open que abre o ficheiro em causa
    //com as devidas permissoes e flags
    int fd = open(file, O_CREAT | O_WRONLY | O_RDONLY, 0666);

    while(1){

                // inteiro para determinar se o buffer termina ou nao com \n
                int has_n = 0;
                // percorre o buffer todo e se este tiver
                // um \n coloca o inteiro has_n a 1
                for(i =0 ; i<sizeof(ps); i++){
                    if(ps[i]=='\n'){
                        has_n = 1;
                        break;
                }}

            //Se a linha comeca com $
```

```
        if(buf[1] == '$'&& signal==0 && strlen(buf+n)-2>0){
            //coloca o buffer em ps
            char *ps = buf;
            // avanÃ§a, de maneir a ignorar o $
            ps = ps+2;
            int i;

            //Mete um \n caso o utilizador se esqueÃ§a de fazer
            if(has_n==0) write(fd, "\n", 1);
            //para escrever no ficheiro, sabemos que a escrita dos outputs
            // tem de ser delimitada por >>> <<<
            write(fd, ">>>\n", 4);
            // executa o comando em causa
            executeCommand(ps, fd);
            write(fd, "<<<\n", 4);
        }
```

Para executar cada comando, foi feita a função **executeCommand**.

Esta função é basicamente um interpretador de comandos que recebe o comando e os argumentos que é suposto executar.

```
void executeCommand(char* command, int fildes){
    int n, i, pid;
    char buffer[4096];
    // apontador para apontador onde vamos guardar os
    // comandos e os argumentos a executar
    char* executar[512];

    //buffer s para onde serÃ¡ enviado o comando a executar
    char* s;
    // criar um pipe (um para escrita e outro para leitura)
    int link[2];
    pid_t pid2;
    pipe(link);

    for(i=0; i!=sizeof(buffer); buffer[i++]='\0');

    // se existir mesmo um comando para executar
    if((n=strlen(command+1)) >0 ){

        // parte "command" com o delimitador " \n\0" e coloca em s
        s = strtok(command, " \n\0");

        for(i=0; s!=NULL; i++){

            // aloca espaÃ§o em memoria num ponteiro
            // de tipo char com espaÃ§o n (comprimento do comando)
            // vezes o tamanho de um caracter
            executar[i] = (char*) malloc(n*sizeof(char));
            // copia o comando/argumento que esta em s
            // para "executar"
            strcpy(executar[i], s);
            // para avanÃ§ar para o proximo argumento da funcao
            s=strtok(NULL, " \n\0");
        }
        // coloca o ultimo elemento a NULL
        executar[i]=NULL;
```

```
            // Como e necessario fazer um execvp para executar
            // os programas e necessario um fork e executar o mesmo
            // no processo filho pois a seguir ao exec vem um exit
            // que mata o processo
            pid = fork();
            if(pid==0){
                // e feito um dup2 de maneira a duplicar
                // o descritor de ficheiro que esta a efetuar a
                // escrita no pipe para 1. ou seja,
                // o que se escreveria no standard out (1) vai
                // se escrever em link[1]
                dup2(link[1], 1);
                // fecha se o descritor de leitura no filho pois este
                // apenas vai escrever
                close(link[0]);

                // faz o execvp do que esta em executar[o], com os
                // argumentos passados
                execvp(executar[0], executar);

                // // para sair do processo filho caso o exec nao funcione
                _exit(-1);

            }else  {
                     close(link[1]);
                     while(1){

                         //aqui efectua a escrita no ficheiro
                         size_t n = sprintf(buffer, "␣");
                         ssize_t nb = read(link[0], buffer+n, sizeof(buffer)-n);
                         write(fildes, buffer+n, nb);
                         if(nb==0) break;
                         int i;
                         for(i = 0; i<nb; i++){
                             if(buffer[i]=='\n'){
                                 buffer[i] = '␣';
                             }
                         }

                         for(i=0; i!=sizeof(buffer); buffer[i++]='\0');
                     }

                wait(NULL);
                }
            }
        }
        numOut +=1;
}
```

```
NoteBook SO

Pathname da diretoria atual:
$ pwd
Agora podemos listar os ficheiros presentes nesta diretoria:
$ ls
O ls pode ter argumentos como o "-l" para listar e o "-t" para ordenar por data de modificação:
$ ls -l -t


```

Figura 3.1: **Notebook antes de ser processado**

```
NoteBook SO

Pathname da diretoria atual:
$ pwd
>>>
/Users/alfredo/Desktop/TPSO
<<<
Agora podemos listar os ficheiros presentes nesta diretoria:
$ ls
>>>
Makefile
exemplo.nb
file.txt
notebook.nb
notebook.nb.kbp
processa
processa.c
<<<
O ls pode ter argumentos como o "-l" para listar e o "-t" para ordenar por data de modificação:
$ ls -l -t

>>>
total 88
-rw-r--r--  1 alfredo  staff    350  2 Jun 00:51 file.txt
-rw-r--r--  1 alfredo  staff    226  2 Jun 00:51 notebook.nb
-rw-r--r--  1 alfredo  staff    240 31 Mai 12:58 notebook.nb.kbp
-rwxr-xr-x  1 alfredo  staff  13876 31 Mai 12:58 processa
-rw-r--r--  1 alfredo  staff   7544 31 Mai 12:57 processa.c
-rw-r--r--@ 1 alfredo  staff    561 31 Mai 12:08 Makefile
-rw-r--r--  1 alfredo  staff    158 30 Mai 11:23 exemplo.nb
<<<
```

Figura 3.2: **Notebook processado**

Comandos que recebem como stdin o resultado anterior(Jonas)

Se começadas por $— o comando deve ter como stdin o resultado do comando anterior.

3.2 Re-processamento de um notebook

O conteudo assim produzido e delimitado, dever ´ a ser ignorado ´ (nao interpretado como comandos).

Para esta parte, foi tido em conta que, uma vez que, na leitura do notebook, todas as linhas que não são comandos são ignoradas, é possível processar um ficheiro que contenha os resultados de uma execução anterior.

Os comandos e resultados desta nova execução são copiados para um ficheiro auxiliar que, posteriormente, irá substituir o ficheiro original. Deste modo, o resultado será um novo notebook apenas com os resultados da última execução.

Quando se vai reprocessar um notebook já anteriormente processado, todas as linhas de texto normal são ignoradas e escritas normalmente, as linhas que são de executar comandos, ou seja, com dolar ou dolar pipe são executdas e o seu output é escrito a seguir, conforme explicado na secção de execução de programas, e todas as linhas entre ¿¿¿ e ¡¡¡, que são outputs de comandos processados da vez anterior, são ignorados e não são escritos no ficheiro.

Tal é possivel através de um sinal que, na leitura do ficheiro, quando deteta que este tem uma linha começada por »¿¿"é colocado a 1 e quando termina o output («¡¡") é colocada a zero. Assim, o processador sabe se deve escrever no ficheiro ou não (util também para não executar comandos dentro de outputs, por exemplo se for feito o comando cat de um notebook processado.

```
int signal = 0;
if(buf[1] == '>' && buf[2] == '>' &&) buf[3] == '>'{
                signal = 1;
                }

            if(signal==0){
            write(fd, buf+n, nb);
            }
else if(buf[1]== '<' && buf[1] == '<' && buf[1] == '<'){
                signal =0;
            }

             if(buf[1] == '$'&& signal==0 && strlen(buf+n)-2>0){
```

No exemplo seguinte, é possivel observar um caso em que, tendo já um notebook processado, muda-se um comando para outro (ls para pwd) e apaga-se o comando sort -r.

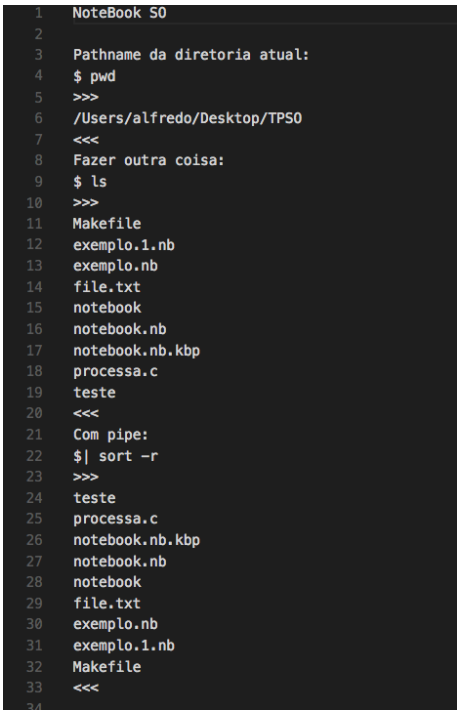

Figura 3.3: **Notebook processado**

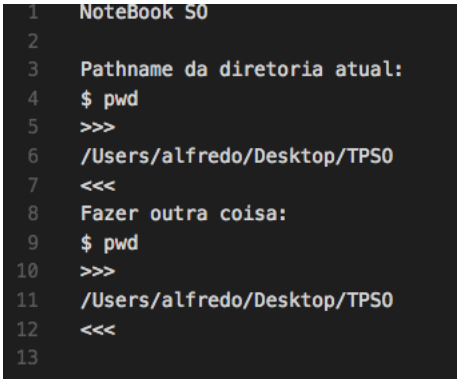

Figura 3.4: **Notebook re-processado**

3.3 Deteção de erros e interrupção da execução

Como referido na introdução, o sistema deverá ser capaz de deteção erros e interrupções de execução e preservar o estado inicial do ficheiro.

Começando pelas interrupções de execução, um utilizador é sempre capaz de interromper uma execução, seja porque se enganou ou porque a execução está a demorar demasiado tempo e assume-se que algo está errado.

Quer seja por uma destas razões ou outra que o utilizador levante como justa, quando se interrompe uma execução com recurso a um CTRL-C, o sistema recebe um sinal de que a execução foi interrompida e executa uma função designada `preservaFicheiro`. Esta função tem o simples objetivo de alterar o valor de uma variável global designada `semErros`, inicialmente com o valor 1, para 0.

Esta alteração terá efeitos na execução de comandos, uma vez que estes só são executados quando o valor da variável se encontra a 1. Não interessa de todo, continuarmos a execução de x comandos quando o primeiro foi interrompido.

Caso a execução de um comando não seja bem sucedida, ou seja, escreva algo no *stderr*, o sistema é capaz de identificar essa situação, uma vez que um processo terá um valor de saída diferente de 0. Quando isso acontece, as condições de paragem são conseguidas e é executada, novamente, a função `preservaFicheiro`, terminando assim a execução de comandos.

Posto isto, na função *main*, verfica-se se o valor da variável global continua a 1, onde o notebook será alterado caso se confirme ou manter-se-á inalterado caso o valor seja 0.

3.4 Execução de comandos anteriores

Na leitura do ficheiro, caso encontre um **$|**, guarda o comando que vem no seu seguimento e usa a função ***executePrev***. Esta função, com ajuda da variável global numOut (que indica o número de ficheiros auxiliares com o output das funções), adiciona ao comando o ficheiro com onde estão os resultados da função anterior (isto porque os ficheiros têm como nome o seu número seguido de .txt, como 1.txt) e envia o comando completo para a função ***executeCommand***.

```
void executePrev(char* command, int fildes){
        char pos[20];
        sprintf(pos, "teste/%d", numOut-1);

        strcat(pos,".txt");

        strcat(command, pos);

        executeCommand(command, fildes);
}
```

A partir daqui o comando é executado normalmente e o seu output é inserido num ficheiro auxiliar e no resultado final.

Infelizmente a parte seguinte não foi implementada pois causava um erro, por isso a parte seguinte contém apenas código não incluido no notebook.

Caso se depare com **$*n*|** o precesso é muito semelhante exceto que ele irá também guardar num array todos os digitos até ao | e envia o resultado do atoi

desse array para o ***executePrev*** que irá buscar o ficheiro (numOut-valor).txt para adicionar ao comando e enviar para o ***executeCommand***.

```
void executePrev(char* command, int fildes, int valor){
        char pos[20];
        sprintf(pos, "teste/%d", numOut-valor);

        strcat(pos,".txt");

        strcat(command, pos);

        executeCommand(command, fildes);
}
```

Capítulo 4

Testes

Para comprovar que o desenvolvimento do sistema foi realizado com sucesso, procedeu-se à realização de testes onde será possível verificar se o processamento dos comandos é feito de forma correta. Assim, seguem uma série de testes realizados, juntamente com uma explicação caso necessário.

Neste primeiro teste, fazemos a listagem dos ficheiros que estão na diretoria atual, seguido de um *sort*, apresentando de seguida o *path* para a diretoria atual.

```
$ ls
$| sort
$ pwd
```

```
$ ls
>>>
Makefile
a.out
exemplo.1.nb
exemplo.nb
file.txt
notebook
notebook.nb
notebook.nb.kbp
processa.c
teste
<<<
$| sort
>>>
Makefile
a.out
exemplo.1.nb
exemplo.nb
file.txt
notebook
notebook.nb
notebook.nb.kbp
processa.c
teste
<<<
$ pwd
>>>
/Users/Francisco/TPSO
```

Neste segundo teste, procuramos testar se a ocorrência de um comando inválido mantém o estado inicial do notebook. Aqui fazemos uma listagem mais pormenorizada seguida de um comando inexistente.

```
$ ls
$ vinteASO eEsteAno
```

```
$ ls
$ vinteASO eEsteAno
```

Tomando como ponto de partida o ficheiro do primeiro teste, desta vez já processado, modificamos apenas um comando e podemos verificar a adaptabilidade a re-processamentos, com presença de um comando diferente.

```
...
processa.c
teste
<<<
$| head -1
>>>
/Users/Francisco/TPSO
<<<
```

```
    ...
processa.c
teste
<<<
$| head -1
>>>
Makefile
<<<
```

Se por acaso, pegando no mesmo ficheiro, colocássemos um comando inexistente, novamente o estado inicial seria conservado.

```
...
processa.c
teste
<<<
$| heade -1
>>>
/Users/Francisco/TPSO
<<<
```

```
    ...
processa.c
teste
<<<
$| heade -1
>>>
/Users/Francisco/TPSO
<<<
```

Estando o nosso sistema capaz de preservar o estado inicial do *notebook* caso interrompêssemos o seu processamento, através de um CTRL-C, este mantém, como seria de prever, o seu estado inicial.

```
$ ls
$ sleep 10
$ cd pasta
```

```
$ ls
$ sleep 10
$ cd pasta
```

Capítulo 5

Conclusão

Terminada a realização do trabalho prático, o grupo encontra-se em posição para avaliar muito positivamente todo o sistema desenvolvido.

Após um debate inicial sobre como deveria ser feita a implementação para o problema em causa, o desenvolvimento deste foi atingido com sucesso e serviu de consolidação dos conhecimentos adquiridos ao longo do semestre, nas aulas práticas e teóricas.