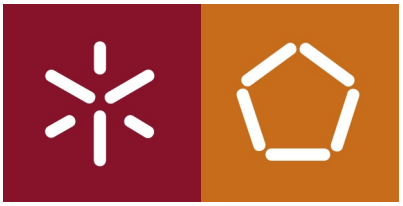

UNIVERSIDADE DO MINHO

Processamento de Notebooks

Sistemas Operativos
Ano letivo 2017/2018

Grupo 36
A71509 Cláudia Marques
A73236 Humberto Vaz
A72062 Ricardo Lopes

Braga
2 de Junho de 2018

Resumo

`Processamento de Notebooks` é um projeto realizado no âmbito da unidade curricular de Sistemas Operativos do 2º ano do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

O objetivo deste projeto é construir um sistema para processamento de notebooks, que misturam fragmentos de código, resultados de execução e documentação.

O processador de *notebooks* é um comando que ao receber um nome de ficheiro, executa os comandos nele embebidos. Ou seja, sempre que encontra um $, executa o comando que se encontra a seguir e imprime o *output* do programa.

Ao longo deste relatório, serão abordados todos os objetivos propostos no enunciado.

Por fim, serão demonstrados alguns testes feitos ao sistema.

Conteúdo

1 Introdução **3**
1.1 Descrição do Problema 3

2 Desenvolvimento **5**
2.1 Script de Instalação 5
2.2 Estruturas Utilizadas 5
2.2.1 **Estrut** 5
2.2.2 **Command** 5
2.3 Processador de Notebooks 6
2.3.1 Execução de Comandos simples 8
2.3.2 Execução de Comandos em pipeline 8
2.3.3 Re-processamento de um notebook 9
2.3.4 Deteção de erros e interrupção de execução 10
2.3.5 Acesso a resultados de comandos anteriores arbitrários . 10

3 Testes **13**

4 Conclusão **16**

Capítulo 1

Introdução

1.1 Descrição do Problema

Tal como referido anteriormente, este trabalho tem como objetivo criar um sistema para processamento de *notebooks*, implementado em C, recorrendo ás primitivas do sistema operativo e os programos utilitários Unix indicados.

O sistema implementado deverá ser capaz de, ao receber um nome de um ficheiro, interpretar as linhas começadas por $ como comandos que serão executados, sendo o resultado produzido inserido imediatamente a seguir, delimitado por $>>>$ *e* $<<<$.

As linhas começadas por $| executam comandos que têm como *stdin* o resultado do comando anterior.

Por exemplo, um ficheiro `exemplo.nb` com o conteúdo:

```
Este comando lista os ficheiros:
$ ls
Agora podemos ordenar estes ficheiros:
$| sort
E escolher o primeiro:
$| head -1
```

O ficheiro do notebook ficará com um conteúdo semelhante ao seguinte:

```
Este comando lista os ficheiros:
$ ls
>>>
colher.c
ananas.out
banana.c
<<<
Agora podemos ordenar estes ficheiros:
$| sort
```

```
>>>
ananas.out
banana.c
colher.c
<<<
E escolher o primeiro:
$| head -1
>>>
ananas.out
```

Um dos requistos do trabalho trata-se da deteção de erros e interrupção da execução. No caso de algum dos comandos não ser executado, não termine com sucesso, ou se este escrever algum conteúdo para o *stderr*, o processamento deve ser anulado, ficando o notebook inalterado. Se o utilizador quiser interromper um processamento em curso, deverá fazer `Ctrl-C` correspondendo a um sinal de de interrupção (SIGINT), ficando o *notebook* inalterado.

Capítulo 2

Desenvolvimento

2.1 Script de Instalação

Foi desenvolvido um script que corre faz build ao projeto e corre os exemplos de teste. Esse ficheiro encontra-se no ficheiro .zip e tem o nome de *run.sh*.

2.2 Estruturas Utilizadas

No começo, foram criadas duas estruturas: *Estrut* e *Command*.

2.2.1 Estrut

Criou-se a estrutura *Estrut*, na qual definiu-se dois campos: o número de linhas do output e um apontador de apontadores *data*. É usada para armazenar o conteúdo do notebook.

```
typedef struct estrutura
{
    int nrlinhas;
    char **data;
} * Estrut;
```

2.2.2 Command

Também se criou a estrutura *Command*, para armazenar a informação dos comandos recebidos, sendo que esta informação se trata do próprio comando e o seu output. Esta estrutura é uma matriz em que cada comando corresponde à primeira coluna de uma linha da matriz e cada argumento (do mesmo comando) corresponde a uma coluna dessa linha.

```
typedef struct command
{
    char **command;
    char **out;
    int nrcoms;
} * Command;
```

2.3 Processador de Notebooks

A estratégia do grupo foi criar dois pipes anónimos, cujos *file descriptors* são `fd` e `fdPipe` para permitir comunicação entre processos.

O primeiro, `fd`, será responsável por conseguir devolver o output gerado pelo exec do processo filho ao processo pai. Já o segundo é usado no caso de comandos com pipe. Utilizámos dois processos. Um que processo vai escrever o output do comando anterior para o fdPipe, enquanto que o outro processo faz o *execvp* do comando atual, mas tendo como input o conteúdo presente no pipe `fdPipe` que foi préviamente escrito tal como referido acima.

Inicialmente, começou-se por usar a função **strtok** para quebrar a string original numa sequência de *tokens* delimitados por espaços, usando uma estrutura temporária, *str*.

```
while ((readFile = fgets(temp, 100, file)) != NULL) {
    temp[strlen(temp) - 1] = '\0';
    linha = strdup(temp);
    char *s = strdup(" ");
    token = strtok(temp, s);
    int i = 0;
    int j = 0;
    while (token != NULL)
    {
        str[i] = strdup(token);
        token = strtok(NULL, s);
        i++;
    }
```

Após o ficheiro ser processado,o programa irá guardar as linhas todas do notebook "não modificadas"em memória na estrutura com o nome `estrut`. No caso de uma linha se tratar de um comando, para além de ser guardada na estrutura referida anteriormente, são guardadas no array da estrutura *command* que tem o nome `commands`.

```
if (str[0] != NULL && str[0][0] != '$')
    {
        if(ignoreLines == 0)
        {
            estrut->data[estrut->nrlinhas++] = strdup(linha);
        }
    }
    else if(str[0] != NULL)
    {

        Command addingCommand;
        addingCommand = initComs();
        for (j = 0; str[j] != NULL; j++)
        {
            addingCommand->command[j] = strdup(str[j]);
        }
        addingCommand->command[j] = NULL;
        commands = addCommand(commands, addingCommand,
            indexCommands++, &sizeCommands);

        estrut->data[estrut->nrlinhas++] = strdup(linha);
    }

    if(strcmp(linha, "<<<")==0)
    {
        ignoreLines = 0;
    }
```

O seguinte código serve para "recolher"o *output* de comandos executados. Para tal, foi utilizado um pipe anónimo, cujo *file descriptor* tem o nome `fd`.

Dentro do ciclo *while*, o output dos comandos é armazenado linha a linha no array `commands`.

```
...
close(fd[WRITE_END]);
int status;
char buf[100];
int counter = 0;
wait(&status);
WEXITSTATUS(status);
if (status == 0) {
      line = 0;
      while ((readPipe = lerLinha(fd[READ_END])) != NULL {
              commands[k]->out[line++] = strdup(readPipe);
      }
      int i;
}
```

```
...
```

2.3.1 Execução de Comandos simples

Tal como foi referido antes, no caso das linhas começarem só por $, são interpretadas como comandos, e o resultado gerado é inserido no ficheiro. Para isso, criou-se um processo filho que será responsável por escrever no *output* esse resultado.

Nas linhas de código estamos a "varrer"os comandos do array da estrutura *command*, com o nome `commands`, e para cada comando: criamos um processo filho (com a invocação da função `fork()`), fazemos uma duplicação do *file descriptor stdout* para o *output* do pipe (`fd[WRITE_END)` para que o comando escreva o seu *output* para o pipe aquando a sua chamada na função `execvp`.

É de notar que fazemos o controlo de erros de execução de comandos ao imprimir-mos uma mensagem de erro para o *stderr* e de seguida lançamos um `exit` com o códifo de erro `-1` para informar o processo pai que o `execvp` correu mal.

```
for (k = 0; k < indexCommands; k++){
   int fd[2];
   int r = pipe(fd);
   if (r < 0)
       perror("Erro no Pipe\n");
   x = fork();
        if (x == 0 && commands[k]->command[0][1] != '|') {
        close(fd[0]);
        dup2(fd[WRITE_END], STDOUT_FILENO);
        execvp(commands[k]->command[1],
            &(commands[k]->command[1]));
        perror("Nao devia imprimir isto\n");
       exit(-1);
       }
}
```

2.3.2 Execução de Comandos em pipeline

No caso das linhas começarem por $| ou $n| (sendo que n trata-se de um número natural), foi a forma definida para definir um pipeline de comandos. Isto é, o output do comando anterior é o input do comando atual (no caso da string se iniciar com $|*comando*) ou o output do n-ésimo comando anterior é o input do comando atual (no caso da string se parecer com $n|*comando*) .

Para resolver este caso, criou-se um processo (filho) que irá escrever no pipe (`fdpipe[WRITE_END]`) de forma a que o *output* do comando anterior seja o *stdin* do processo que vai executar o comando, sendo que este é o processo (pai) que foi criado antes com a seguinte linha: `x = fork();`.

```
x = fork();
...
if (x == 0 && commands[k]->command[0][1] == '|'){
...
   int fdPipe[2];
   pipe(fdPipe);

   int y;
   y = fork();
   if (y == 0) {
       close(fdPipe[0]);
       dup2(fdPipe[WRITE_END], STDOUT_FILENO);
       int l;
       for (l = 0; commands[k - 1]->out[l] != NULL; l++) {
           write(fdPipe[WRITE_END], commands[k - 1]->out[l],
               strlen(commands[k - 1]->out[l]));
           write(fdPipe[WRITE_END], "\n", 1);
       }
       close(fdPipe[WRITE_END]);
       exit(0);
   }
   ...
}
```

2.3.3 Re-processamento de um notebook

De forma a tornar o programa funcional e proporcionar ao utilizador uma ferramenta de que permite "anotar"informação bem como executar comandos *unix*, procedeu-se à implementação da seguinte forma:

```
int ignoreLines = 0;
...
if(strcmp(linha, ">>>")==0)
{
    ignoreLines = 1;
}

...
/* Adicao de comandos ao array "commands" */
...
if(strcmp(linha, "<<<")==0)
{
```

```
    ignoreLines = 0;
}
```

Dado que este código é executado dentro do ciclo while inicial que está a captar linha a linha do ficheiro notebook, conseguimos filtrar apenas o conteúdo diferente de »»"ou ««", marcando o inicio e fim de conteúdo a ignorar com o auxílio de uma flag, *ignoreLines* e desta forma proporcionar ao utilizador um re-preocessamento do seu notebook

2.3.4 Deteção de erros e interrupção de execução

Nos casos de erros, é enviada uma mensagem de erro, que é escrita no *stderr* e é feito *exit(-1)*. No caso de ser um processo filho, o processo pai estará à espera do fim da sua execução e irá verificar o seu exitstatus, que se este for diferente de zero o processo pai irá também fazer *exit(-1)*. Como na nossa implementação de código só estamos a modificar o ficheiro no final da execução de todos os comandos e se todos tiverem corrido com sucesso, estamos a garantir que se caso algum processo intermediário corra de forma errada, o ficheiro permanece inalterado. Por exemplo:

```
...
perror("Error on waiting for child process\n");
exit(-1);
...
wait(&status);
WEXITSTATUS(status);
if (status == 0)
{
    ...
}
else
{
    perror("Error on waiting for child process\n");
    exit(1);
}
```

2.3.5 Acesso a resultados de comandos anteriores arbitrários

De forma a conseguirmos executar um pipeline através de *macros*/atalhos, implementou-se uma solução que visa facilitar ao utilizador uma forma de inserção de comando mediante o histórico de comandos.

Tal como foi brevemente referido anteriormente, caso um utilizador queira introduzir o n-ésimo comando anterior em pipeline com um outro, basta intoduzir da seguinte forma no ficheiro de notebook:

```
$n|comando
```

Sendo que n se trata de um número natural válido mediante o histórico de comandos e comando se trata de um comando unix válido também.

Esta solução foi implementada da seguinte forma:

```
if (x == 0 && (commands[k]->command[0][1] == '|' ||
    isdigit(commands[k]->command[0][1])))
{
...
    while (isdigit(commands[k]->command[0][index]) != 0)
    {
        tempNumber[index - 1] = commands[k]->command[0][index];
        hasDigit = 1;
        index++;
    }
```

No código acima é possível verificar se se trata de um comando com pipeline de um output de um comando passado (isto é, um número) e guardá-lo numa variável temporária, `tempNumber` que se trata de um array de carateres.

```
y = fork();

if (y == 0)
{
...
    if (hasDigit == 1)
    {
        previousCommand = atoi(tempNumber);
        if (previousCommand > 0 && previousCommand <= indexCommands)
            commandIndex = commandIndex - previousCommand;
        else
        {
        perror("Comando invalido");
        exit(-1);
        }
    }
    else
    {
        commandIndex = commandIndex - previousCommand;
    }
...
}
```

No caso de este se tratar de um comando "macro"como referimos antes, este irá executar o código correspondente à condição `if (hasDigit == 1)` e guardar o resultado da conversão do array de carateres `tempNumber` na variável inteira `previousCommand` através da função `atoi(tempNumber)`.

No caso contrário, trata-se da execução em pipeline do comando anterior (correspondente ao $1) com um comando atual (inserido de seguida) e irá executar o comando else.

É de referir que a variável `previousCommand` quando declarada, é inicializada a com o número 1 de forma a que a execução com o pipeline do comando anterior funcione.

Capítulo 3

Testes

Com o intuíto de se tornar fácil de analisar o nosso programa, iremos colocar uma série de testes executados na máquina pessoal de um de nós.

Mediante a inserção de num ficheiro por exemplo com o nome notebook.nb com a seguinte informação:

```
Este comando lista os ficheiros e mostra informação
$ ls -la
Este conta as palavras
$| wc -c
Este mostra os processos
$ ps
Este ordena o resultado do 3º último comando
$3| sort
```

Após invocar o programa na shell da seguinte forma:

```
./notebook notebook.nb
```

Iremos obter um output semelhante ao seguinte:

```
Este comando lista os ficheiros e mostra informação
$ ls -la
>>>
total 304
drwxr-xr-x  16 humbertovaz  staff    544 Jun  2 21:17 .
drwxr-xr-x@  5 humbertovaz  staff    170 May 23 14:32 ..
-rw-r--r--@  1 humbertovaz  staff   6148 Jun  2 18:04 .DS_Store
drwxr-xr-x  15 humbertovaz  staff    510 Jun  2 21:17 .git
-rw-r--r--   1 humbertovaz  staff     19 May 27 23:40 .gitignore
drwxr-xr-x   3 humbertovaz  staff    102 Jun  2 15:38 .vscode
-rw-r--r--   1 humbertovaz  staff    518 May 31 14:54 README.md
-rwxr-xr-x   1 humbertovaz  staff  14472 Jun  2 16:59 a.out
```

```
drwxr-xr-x@  3 humbertovaz  staff    102 Jun  2 15:38 a.out.dSYM
-rw-r--r--@  1 humbertovaz  staff  74687 Apr 29 19:03 enunciado-so-2017-18.pdf
-rwxr-xr-x   1 humbertovaz  staff  13924 Jun  2 21:13 leNotebook
-rw-r--r--   1 humbertovaz  staff   8227 Jun  2 21:12 leNotebook.c
-rw-r--r--@  1 humbertovaz  staff    183 Jun  2 21:17 notebook
-rwxr-xr-x   1 humbertovaz  staff     17 May 16 16:42 run.sh
-rw-r--r--   1 humbertovaz  staff   2375 Jun  2 21:17 teste.nb
-rw-r--r--   1 humbertovaz  staff   2720 Jun  2 17:04 teste1.nb
<<<
Este conta as palavras
$| wc -c
>>>
    1024
<<<
Este mostra os processos
$ ps
>>>
  PID TTY           TIME CMD
 3471 ttys000    0:00.67 /bin/zsh -l
  637 ttys001    0:00.04 /Applications/iTerm (3.0.13).app/Contents/MacOS/iTerm2 --serv
umbertovaz
  639 ttys001    0:02.70 -zsh
 9453 ttys001    0:00.00 ./leNotebook notebook
<<<
Este ordena o resultado do 3º último comando
$3| sort
>>>
-rw-r--r--   1 humbertovaz  staff     19 May 27 23:40 .gitignore
-rw-r--r--   1 humbertovaz  staff    518 May 31 14:54 README.md
-rw-r--r--   1 humbertovaz  staff   2375 Jun  2 21:17 teste.nb
-rw-r--r--   1 humbertovaz  staff   2720 Jun  2 17:04 teste1.nb
-rw-r--r--   1 humbertovaz  staff   8227 Jun  2 21:12 leNotebook.c
-rw-r--r--@  1 humbertovaz  staff    183 Jun  2 21:17 notebook
-rw-r--r--@  1 humbertovaz  staff   6148 Jun  2 18:04 .DS_Store
-rw-r--r--@  1 humbertovaz  staff  74687 Apr 29 19:03 enunciado-so-2017-18.pdf
-rwxr-xr-x   1 humbertovaz  staff     17 May 16 16:42 run.sh
-rwxr-xr-x   1 humbertovaz  staff  13924 Jun  2 21:13 leNotebook
-rwxr-xr-x   1 humbertovaz  staff  14472 Jun  2 16:59 a.out
drwxr-xr-x   3 humbertovaz  staff    102 Jun  2 15:38 .vscode
drwxr-xr-x  15 humbertovaz  staff    510 Jun  2 21:17 .git
drwxr-xr-x  16 humbertovaz  staff    544 Jun  2 21:17 .
drwxr-xr-x@  3 humbertovaz  staff    102 Jun  2 15:38 a.out.dSYM
drwxr-xr-x@  5 humbertovaz  staff    170 May 23 14:32 ..
total 304
<<<
```

Outro exemplo de teste:

```
Este comando mostra os processos:
$ ps
Este mostra a pasta atual:
$ pwd
Este ordena os processos:
$2| sort
```

De forma análoga, ao introduzirmos na shell o comando referido acima, obtemos:

```
Este comando mostra os processos:
$ ps
>>>
  PID TTY           TIME CMD
 3471 ttys000    0:00.67 /bin/zsh -l
  637 ttys001    0:00.04 /Applications/iTerm (3.0.13).app/Contents/MacOS/iTerm2 --serv
umbertovaz
  639 ttys001    0:03.00 -zsh
11605 ttys001    0:00.00 ./leNotebook teste.nb
<<<
Este mostra a pasta atual:
$ pwd
>>>
/Users/humbertovaz/Desktop/SO/trabalhopratico/OperatingSystems
<<<
Este ordena os processos:
$2| sort
>>>
  637 ttys001    0:00.04 /Applications/iTerm (3.0.13).app/Contents/MacOS/iTerm2 --serv
  639 ttys001    0:03.00 -zsh
  PID TTY           TIME CMD
 3471 ttys000    0:00.67 /bin/zsh -l
11605 ttys001    0:00.00 ./leNotebook teste.nb
umbertovaz
<<<
```

Capítulo 4

Conclusão

Neste trabalho, o principal objetivo era a utilização da linguagem de programação C, para a criação de um programa de *Processamento de Notebooks*.

Em relação ao resultado final do trabalho, o grupo foi capaz de alcançar todos os objetivos propostos, e ainda satisfazer, de forma correta, todas as funcionalidades básicas enunciadas, sendo que apenas não foi implementada a execução de conjuntos de comandos no que diz respeito as funcionalidades avançadas.