Sistemas Operativos

LCC - 2º ano

Trabalho Prático
Processamento de Notebooks
Relatório

João Dias
66850

Nelson Sá
70925

Rafael Silva
ae4282

2 de Junho de 2018

Conteúdo

Capítulo 1

Introdução

Este trabalho prático tem como objetivos consolidar e avaliar os conhecimentos lecionados nas aulas práticas de Sistemas Operativos. Pretende-se construir um sistema para processamento de notebooks, que misturam fragmentos de código, resultados da execução, e documentação. Neste contexto, um notebook é um ficheiro de texto que depois de processado é modificado de modo a incorporar resultados da execução de código ou comandos nele embebidos.

Capítulo 2

Análise e Especificação

2.1 Processador de Notebooks

O processador de notebooks é um programa que dado um nome de ficheiro, interpreta-o, executa os comandos nele embebidos, e acrescenta o resultado da execução dos comandos ao ficheiro. Assim, neste trabalho basta abrir um terminal na diretoria do programa e correr o mesmo (./notebook "nomeDoFicheiro") ou invocar a make.

2.2 Funcionalidades Básicas

Na main, o programa começa por abrir o ficheiro de input (nomeDoficheiro.nb.), os ficheiros temporarios "error.txt"(para onde serão enviados as mensagens de erro), "temp.txt"(onde será guardado, temporariamente, o output da execução do programa) e "lastOutput.txt"(onde é guardado o output do ultimo comando executado). Caso estes não existam, são criados. Caso já existam são truncados. De seguida, começa a ler o ficheiro de input linha a linha (dentro de um ciclo while, até ao end of file) e converte a string que guarda a linha em um array de strings (palavras). Depois verifica o conteúdo da primeira palavra processando de maneira diferente consoante seja um output de um processamento anterior do programa, comentários ou comandos (processando de maneira diferente consoante seja comando isolado ou ligado a outputs anteriores).

```
int main(int argc, char* argv[]){
  int fd = open(argv[1], O_RDWR, 0666);
  int dir;
  dir = mkdir("outputs",0777);
  int fd_error = open("error.txt", O_CREAT| O_RDWR| O_TRUNC, 0666);
  dup2(fd_error,2);
```

```
  close(fd_error);
  int fd1 = open("temp.txt", O_CREAT| O_TRUNC, 0666);
  close(fd1);
  int fd2 = open("outputs/lastOutput", O_CREAT| O_TRUNC| O_RDWR,0666);
  close(fd2);
  char buf[512];
  int n;
  int cmdsLidos=0;
  int flag = 1;
  while((n = readln(fd,buf,1)) > 0){...}
```

2.2.1 Execução de Programas

Quando o programa encontra uma linha iniciada por $ (indicando que é linha de comandos) incrementa uma variável que guarda o numero de comandos e chama a função processCmds().

```
...
if(!strcmp(cmds[0],"$")){
      cmdsLidos++;
      processCmds(cmds,cmdsLidos);
}
...
```

A função processCmds() recebe como argumentos o array de strings que contem o comando e seus argumentos e o numero de comandos lidos até ao momento (ele incluído). Esta função cria um processo filho que redireciona o seu file descriptor de saída para o seu processo pai e executa o comando e seus possíveis argumentos, passados no array de strings a partir da segunda posição. O processo pai vai acrescentar o resultado da execução no ficheiro "temp.txt"entre “»>” e “«<” ,vai ainda copiar o resultado para o "lastOutput.txt"e criar um ficheiro (em que o nome é o numero de comandos lidos e executados até ao momento, ou seja, o numero de ordem deste comando) onde se vai guardar o resultado até ao fim da execução do programa.

```
int processCmds(char** cmds,int cmdsLidos){
  int fd2[2];
  pipe(fd2);
  if(!fork()){
    dup2(fd2[1],1);
    close(fd2[1]);
    close(fd2[0]);
    if(execvp(cmds[1],&cmds[1]) == -1) _exit(1);
  }
  int status;
  wait(&status);
```

```
  if(WEXITSTATUS(status) == 1){write(1,"ERROR\n",6); exit(0);}
  close(fd2[1]);
  int fd = open("temp.txt", O_WRONLY| O_APPEND, 0666);
  int fd3 = open("outputs/lastOutput", O_RDWR, 0666);
  char name[16];
  sprintf(name,"outputs/%d",cmdsLidos);
  int fd4 = open(name,O_CREAT| O_RDWR| O_TRUNC, 0666);
  write(fd,">>>\n",4);
  char buf;
  while(read(fd2[0],&buf,1)>0){
    write(fd,&buf,1);
    write(fd3,&buf,1);
    write(fd4,&buf,1);
  }
  write(fd,"<<<\n",4);
  close(fd2[0]);
  close(fd3);
  close(fd4);
  close(fd);
  return 0;
}
```

2.2.2 Re-processamento de um notebook

Depois de o programa ser executado a primeira vez, o ficheiro de input vai passar a conter os outputs do ultimo processamento, entre "»>" e "«<". Assim, num novo processamento, o programa deve ignorar esse conteúdo. Na função main, dentro do ciclo que lê linha a linha, a iteração só corre caso uma flag esteja a 1, caso contrário avança para a seguinte. Esta flag é alterada quando encontra uma linha com o conteúdo "»>" passando o programa a ignorar todas as linhas até encontrar o conteúdo "«<". Passando o programa a processar "normalmente".

```
...
int flag = 1;
while((n = readln(fd,buf,1)) > 0){
    if(buf[0]=='>') flag=0;
    if(buf[0]=='<'){
      flag =1;
      continue;
    }
    if(flag == 0) continue;
...
```

2.2.3 Detecção de erros e interrupção da execução

No inicio do programa, a saída de erros é redirecionada para um ficheiro error.txt.

```
...
int fd_error = open("error.txt", O_CREAT| O_RDWR| O_TRUNC, 0666);
dup2(fd_error,2);
close(fd_error);
...
```

Assim, antes de copiar o conteúdo do ficheiro temporário "temp.txt"para o de input, se o ficheiro de erros estiver vazio pode-se avançar com essa cópia.

```
...
if(isEmpty("error.txt")){
    fd = open("notebook.nb", O_WRONLY| O_TRUNC, 0666);
    fd1 = open("temp.txt", O_RDONLY, 0666);
    while((n = readln(fd1,buf,1)) > 0){
        write(fd,buf,n);
    }
    write(1,"SUCCESSFUL EXECUTION\n",21);
}else write(1,"EXECUTION ERROR\n",16);
close(fd);
close(fd1);
...
```

Ainda durante a execução, se ocorrer um erro ao executar os comandos, o programa encerra sem alterar o ficheiro de input inicial.

```
...
    if(execvp(cmds[1],&cmds[1]) == -1) _exit(2);
  }
  int status;
  wait(&status);
  if(WEXITSTATUS(status) == 2) {write(1,"ERROR\n",6); exit(0);}
...
```

2.3 Funcionalidades Avançadas

No enunciado do trabalho prático foi pedida a implementação de duas funcionalidades avançadas. O acesso a resultados de comandos anteriores arbitrários e a execução de conjuntos de comandos. Neste trabalho foi implementada a primeira funcionalidade avançada, que vai ser explicada em seguida.

2.3.1 Acesso a resultados de comandos anteriores arbitrários

Esta funcionalidade deve permitir que os comandos possam ser generalizados de forma a terem como stdin o resultado n-ésimo comando anterior, (por exemplo, $4| para se referir ao quarto comando anterior, sendo $1| equivalente a $|) e evitando a re-execução dos comandos.

Este programa, quando encontra os carateres '$' e '|' e um inteiro entre eles (ou seja, na posição 1 do array da string da palavra "$n|") faz um calculo para verificar que o numero de ordem do comando anterior do qual queremos o resultado não é inferior a 1 (ou seja, que existe). De seguida, chama a função processNumPipedCmds(), que recebe como argumentos o array de strings com o comando e seus argumentos, o numero de comandos lidos até ao momento (ele incluído) e o numero de ordem do comando do qual queremos o resultado.

```
...
if(cmds[0][0]=='$' && isdigit(cmds[0][1]) && cmds[0][2] == '|'){
      cmdsLidos++;
      int cmdAler = cmdsLidos-atoi(&cmds[0][1]);
      if(cmdAler < 1){
        write(1,"ERROR\n",6);
        write(2,"ERRO: PUSH_BACK DEMASIADO GRANDE\n",32);
        exit(0);
      }
      processNumPipedCmds(cmds,cmdsLidos,cmdAler);
    }
...
```

Esta função cria um processo filho que redireciona o seu file descriptor de saída para o seu processo pai, redireciona o file descriptor de entrada para o ficheiro com nome igual ao numero de ordem do comando do qual queremos o resultado e executa o comando e seus possíveis argumentos, passados no array de strings a partir da segunda posição. O processo pai vai acrescentar o resultado da execução no ficheiro "temp.txt"entre "»>" e "«<" ,vai ainda copiar o resultado para o "lastOutput.txt"e criar um ficheiro (em que o nome é o numero de comandos lidos e executados até ao momento, ou seja, o numero de ordem deste comando) onde se vai guardar o resultado até ao fim da execução do programa.

```
int processNumPipedCmds(char** cmds, int cmdsLidos, int cmdAler){
  int fd2[2];
  pipe(fd2);
  if(!fork()){
    dup2(fd2[1],1);
    close(fd2[1]);
    close(fd2[0]);
    char name[16];
    sprintf(name,"outputs/%d",cmdAler);
```

```
    int fd = open(name,O_RDONLY);
    dup2(fd,0);
    close(fd);
    if(execvp(cmds[1],&cmds[1]) == -1) _exit(2);
  }
  int status;
  wait(&status);
  if(WEXITSTATUS(status) == 2) {write(1,"ERROR\n",6); exit(0);}
  close(fd2[1]);
  int fd = open("temp.txt", O_WRONLY| O_APPEND, 0666);
  int fd3 = open("outputs/lastOutput", O_RDWR| O_TRUNC, 0666);
  char name[16];
  sprintf(name,"outputs/%d",cmdsLidos);
  int fd4 = open(name,O_CREAT| O_RDWR| O_TRUNC, 0666);
  write(fd,">>>\n",4);
  char buf;
  while(read(fd2[0],&buf,1)){
    write(fd,&buf,1);
    write(fd3,&buf,1);
    write(fd4,&buf,1);
  }
  close(fd2[0]);
  close(fd3);
  close(fd4);
  write(fd,"<<<\n",4);
  close(fd);
  return 0;
}
```

Capítulo 3

Testes realizados e Resultados

3.1 Teste 1

3.1.1 Input

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

3.1.2 Output

```
Este comando lista os ficheiros:
$ ls
>>>
enunciado-so-2017-18.pdf
error.txt
makefile
notebook
notebook.c
notebook.h
notebook.nb
outputs
relatorio_so.odt
relatorio_so.pdf
temp.txt
<<<
```

```
Agora podemos ordenar estes ficheiros:
$| sort
>>>
enunciado-so-2017-18.pdf
error.txt
makefile
notebook
notebook.c
notebook.h
notebook.nb
outputs
relatorio_so.odt
relatorio_so.pdf
temp.txt
<<<
E escolher o primeiro:
$| head -1
>>>
enunciado-so-2017-18.pdf
<<<
```

3.2 Teste 2

3.2.1 Input

```
Este comando lista os ficheiros:
$ ls
>>>
enunciado-so-2017-18.pdf
error.txt
makefile
notebook
notebook.c
notebook.h
notebook.nb
outputs
relatorio_so.odt
relatorio_so.pdf
temp.txt
<<<
Agora podemos ordenar estes ficheiros:
$| ls -l
>>>
enunciado-so-2017-18.pdf
error.txt
makefile
```

```
notebook
notebook.c
notebook.h
notebook.nb
outputs
relatorio_so.odt
relatorio_so.pdf
temp.txt
<<<
E escolher o primeiro:
$| head -1
>>>
enunciado-so-2017-18.pdf
<<<
```

3.2.2 Output

```
Este comando lista os ficheiros:
$ ls
>>>
enunciado-so-2017-18.pdf
error.txt
makefile
notebook
notebook.c
notebook.h
notebook.nb
outputs
relatorio_so.odt
relatorio_so.pdf
temp.txt
<<<
Agora podemos ordenar estes ficheiros:
$| ls -l
>>>
total 240
-rw-rw-r-- 1 rafa rafa 74687 abr 26 12:21 enunciado-so-2017-18.pdf
-rw-rw-r-- 1 rafa rafa  0 jun  2 17:07 error.txt
-rw-rw-r-- 1 rafa rafa 161 jun 1 11:49 makefile
-rwxrwxr-x 1 rafa rafa 17584 jun 2 17:07 notebook
-rw-r--r-- 1 rafa rafa 5643 jun 2 16:48 notebook.c
-rw-r--r-- 1 rafa rafa 585 jun 1 14:18 notebook.h
-rw-r--r-- 1 rafa rafa 445 jun 2 17:07 notebook.nb
drwxrwxr-x 2 rafa rafa 4096 jun 2 17:07 outputs
-rw-rw-r-- 1 rafa rafa 37735 jun 1 14:09 relatorio_so.odt
-rw-rw-r-- 1 rafa rafa 77509 jun 1 14:09 relatorio_so.pdf
-rw-rw-r-- 1 rafa rafa 232 jun 2 17:07 temp.txt
```

```
<<<
E escolher o primeiro:
$| head -1
>>>
total 240
<<<
```

3.3 Teste 3

3.3.1 Input

```
Este comando lista os ficheiros:
$ ls
Mostrar a diretoria:
$ pwd
E escolher o primeiro do ls:
$2| head -1
```

3.3.2 Output

```
Este comando lista os ficheiros:
$ ls
>>>
enunciado-so-2017-18.pdf
error.txt
makefile
notebook
notebook.c
notebook.h
notebook.nb
outputs
relatorio_so.odt
relatorio_so.pdf
temp.txt
<<<
Mostrar a diretoria:
$ pwd
>>>
/home/rafa/Documentos/SO/Trabalho
<<<
E escolher o primeiro do ls:
$2| head -1
>>>
```

```
enunciado-so-2017-18.pdf
<<<
```

Capítulo 4

Conclusão

Ao longo do trabalho prático conseguimos aplicar conhecimentos adquiridos nas aulas práticas da disciplina apesar de não ter sido concluída a segunda funcionalidade avançada pedida no enunciado. Desta forma, o processamento do notebook permite várias operações à exceção da execução de conjuntos de comandos. Assim, podemos dizer que o trabalho prático foi concluído com relativo sucesso.

Capítulo 5

Apêndice

5.1 Outras Funções

5.1.1 isEmpty()

A função isEmpty() recebe como argumento o nome ou caminho de um ficheiro e simplesmente verifica se tem algum conteúdo. Se não tem conteúdo retorna o inteiro 1. Caso contrário retorna o inteiro 0.

```
int isEmpty(char* fileName){
  int fd = open(fileName,O_RDONLY,0666);
  int flag=1;
  char buf;
  while(read(fd,&buf,1)>0){
    flag=0;
    break;
  }
  return flag;
}
```

5.1.2 strcopy()

A função strcopy() recebe como argumentos duas strings e um inteiro 'n' e copia 'n' bits da primeira string para a segunda, retornando quantas palavras existem na string.

```
int strcopy(char* str, char* str2, int bits){
  int count = 1;
  while(bits>=0){
```

```
    if(str[bits] == ' ') count++;
    str2[bits] = str[bits];
    bits--;
  }
  return count;
}
```

5.1.3 split()

A função split() recebe como argumentos uma string e um array de arrays e copia todas as palavras da string para o array exceto os espaços em branco. Retorna o numero de palavras no array.

5.1.4 strToArray()

A função strToArray() recebe uma string e um array de arrays e usando as funções strcopy() e split() processa a informação da string de forma a que o conteúdo do array sejam strings (ou palavras) que vão corresponder a comandos e seus argumentos.

```
int strToArray(char* buf, char** cmds, int n){
  //passa o conteudo da string do 'buf' para um array de
      strings/palavras (cmds)
  buf[n-1] = '\0';
  char linha[n-1];
  int words = strcopy(buf,linha,n-1);
  split(linha,cmds);
  return 0;
}
```