UNIVERSIDADE DO MINHO

LICENCIATURA DE CIÊNCIAS DA COMPUTAÇÃO

SISTEMAS OPERATIVOS

PROCESSAMENTO DE NOTEBOOKS

Pedro Dias (63389) Sérgio Oliveira(62134)

2/06/2018

Índice

Introdução

Este relatório tem como objetivo 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 serem interpretadas pela shell do sistema, resultados de execução das respetivas linhas de código bem como outro tipo de conteúdo tal como documentação relevante.

Relacionado com os temas mencionados, serão apresentados ao longo do relatório fragmentos de código para que haja uma melhor compreensão do assunto em questão.

O relatório encontra-se dividido em várias secções que correspondem aos pontos do enunciado do trabalho, tal como abaixo apresentado.

1. Funcionalidades básicas
   1.1 Execução de programas
   1.2 Re-processamento de um notebook
   1.3 Deteção de erros e interrupção da execução
2. Funcionalidades avançadas
   2.1 Acesso a resultados de comandos anteriores arbitrários
   2.2 Execução de conjuntos de comandos

Desenvolvimento

1.Funcionalidades básicas

1.1 Execução de programas

Neste ponto, o programa deve interpretar as linhas começadas por $ como comandos (programa e argumentos), e se começadas por $| o comando deve ter como *stdin* o resultado do comando anterior. O resultado produzido é então colocado no ficheiro de input e delimitado por “>>>” e “<<<”.

O exemplo escolhido foi o fornecido no enunciado do trabalho.

```
Este comando lista os ficheiros:
$ ls

Agora podemos ordenar estes ficheiros:
$| sort

E escolher o primeiro:
$| head -1

```

Figura 1: Ficheiro “comandos1.nb”.

Para a leitura do ficheiro comandos1.nb e de todos os ficheiros passados como argumento, foi utilizado um ciclo *while* que faz a leitura linha a linha até ao fim do ficheiro.

A cada iteração do ciclo, é enviado para o ficheiro “out.txt” (previamente criado) a linha lida até encontrar o primeiro sinal de $. Após encontrar este primeiro comando, é feito o *parsing* dessa linha para separar o sinal de $ e o comando. Para isto é utilizado a função auxiliar “trim”.

```
//Separa os comandos dos $'s
char * trim(char * s) {
    int l = strlen(s);
    while(isspace(s[l - 1])) --l;
    while(* s && isspace(* s)) ++s, --l;
    return strndup(s, l);
}
```

Figura 2: Função "trim".

Posteriormente é criado um novo ficheiro, “result1.txt” (colocado no diretório “tmp” criado no início do programa), que é utilizado como stdout da execução da system call “execl”.

É criado um *fork* para a execução do “execl” anteriormente descrito e é feita a espera do processo “filho” pelo processo “pai” para que depois seja transferido todo o conteúdo do ficheiro “result1.txt” delimitado por “>>>” e “<<<” para o ficheiro “out.txt”.

```
189             if(strncmp(dollar,line,strlen(dollar))==0 && (running)){
190                 //printf("DOLLAR\n" );
191                 contador++;
192                 contComandos++;
193                 char *b =  trim(line + 2);
194                 printf("comando nº: %d \t %s\n",contador,b);
195
196                 strcatFilename(contador); // ABRIR RESULTN.TXT
197                 result1 =fopen(filename,"wr+");
198
199                 int p=fork();
200                 if(p==0){
201                     dup2(fileno(result1),1); //STDOUT_FILENO
202                     fclose(result1);
203                     execl("/bin/sh", "/bin/sh", "-c", b, NULL);
204                     exit(-1);
205
206                 }
207                 else{
208                     int status;
209                     wait(&status);
210
211                     flagErrorFork=forkError(status,b);
212                     if( flagErrorFork == 0){ // 0 deu erro no fork
213                         break;
214                     }
215
216
217                     fclose(result1);
218
219                     fputs(">>>\n",out);
220                     FILE * result2;
221                     result2 =fopen(filename,"r");
222                     //escrever para out.txt
223                     while ((read2 = getline(&line2, &len2, result2)) != -1){
224                             fputs(line2,out);
225                         }
226                     fputs("<<<\n",out);
227                     }
228             }
```

Figura 3: Parsing do comando "$".

A lógica de execução dos comandos seguidos por $| é a mesma seguida por $ mas com a diferença que esta execução tem como *stdin* o resultado anterior, ou seja, neste caso a execução do comando “sort” vai ter como *stdin* o ficheiro “result1.txt” e como *stdout* o ficheiro “result2.txt”.

```
232             if(strncmp(dollarPipe,line,strlen(dollarPipe))==0 && (running)){
233                 //printf("DOLLAR_PIPE\n" );
234                 char *b =  trim(line + 3);
235                 contador++;
236                 printf("comando nº: %d \t %s\n",contador,b);
237                 strcatFilename(contador);
238                 FILE * resultN;
239                 resultN =fopen(filename,"wr+"); //abre o resultN.txt
240                 int d=fork();
241                 if(d==0){
242                     // ABRIR RESULTN.TXT
243                     strcatFilename(contador-1);
244                     result1 =fopen(filename,"r");
245                     dup2(fileno(result1),0);//STDIN_FILENO
246                     dup2(fileno(resultN),1); //STDOUT_FILENO
247                     fclose(result1);
248                     fclose(resultN);
249                     execl("/bin/sh", "/bin/sh", "-c", b, NULL);
250                     exit(-1);
251                 }
252                 else{
253                     //wait(0);
254                     int status;
255                     wait(&status);
256                     flagErrorFork=forkError(status,b);
257                     if( flagErrorFork == 0){ // 0 deu erro no fork
258                         break;
259                     }
260                     fclose(result1);
261                     fputs(">>>\n",out);
262                     FILE * result2;
263                     result2 =fopen(filename,"r");
264                     //escrever para out.txt
265                     while ((read2 = getline(&line2, &len2, result2)) != -1){
266                             fputs(line2,out);
267                         }
268                     fputs("<<<\n",out);
269                 }
270             }
```

Figura 4: Parsing do comando "$|".

1.2 Re-processamento de um notebook

O re-processamento de um notebook é processado através da função auxiliar "re_processamento". Esta função tem como objetivo criar um novo ficheiro com o nome de "REDO.txt" e colocar no mesmo todas as linhas do ficheiro de input que não estejam entre ">>>" e "<<<" (sinais incluídos), para posteriormente o ciclo *while* descrito na secção 1.1 fazer o processamento dos comandos editados pelo utilizador.

```
105 void re_processamento(char * file){
106     char * line = NULL;
107     size_t bufsize = 32;
108     size_t len=0;
109     ssize_t read=2;
110     FILE * fp;
111     fp = fopen(file, "r");
112     FILE *REDO;
113     REDO = fopen("REDO.txt", "wr+");
114     if (fp == NULL)
115         exit(EXIT_FAILURE);
116     //este ciclo pecorre linha a linha o ficheiro dado como input e coloca
117     //no ficheiro "REDO.txt" todas as linhas que não estejam entre >>> e <<<
118     while ((read = getline(&line, &len, fp)) != -1){
119         if(strncmp(">>>\n",line,strlen(">>>\n"))==0){
120             read = getline(&line, &len, fp);
121             while(strncmp("<<<\n",line,strlen("<<<\n"))!=0)
122                 read = getline(&line, &len, fp);
123         }
124         if(strncmp("<<<\n",line,strlen("<<<\n"))==0){
125             read = getline(&line, &len, fp);
126         }
127         fputs(line,REDO);
128     }
129     fclose(fp);
130     fclose(REDO);
131     rename("REDO.txt",file);
132     if (line) free (line);
133         line = NULL;
134 }
```

Figura 5: Função "re_processamento".

1.3 Deteção de erros e interrupção da execução

A deteção de erros é verificada a cada execução de comandos. Após a criação do *fork* o processo pai fica à espera do processo filho e, se neste ocorrer algum erro na execução do comando, é executada a linha de código “exit(-1)” e o processo pai apanha o sinal de erro, terminando o processamento do ficheiro de input.

```
int p=fork();
if(p==0){
    dup2(fileno(result1),1); //STDOUT_FILENO
    fclose(result1);
    execl("/bin/sh", "/bin/sh", "-c", b, NULL);
    exit(-1);

}
else{
    int status;
    wait(&status);

    flagErrorFork=forkError(status,b);
    if( flagErrorFork == 0){ // 0 deu erro no fork
        break;
    }
```

Figura 6: Execução do "fork".

```
77  int forkError(int status,char *b){
78      int result=1;
79      if(WIFEXITED(status))
80          {
81              if(WEXITSTATUS(status) != 0)
82                  {
83                      printf("ERRO no comando %s\n",b);
84                      printf("fork: %s\n", strerror(errno));
85                      result=0;
86                      }
87                  }
88              else if (WIFSIGNALED(status))
89                  {
90                      printf("terminated because it didn't catch signal number %d\n",
91                             WTERMSIG(status));
92                      result=0;
93                  }
94              else
95                  {
96                  printf("ERRO no comando %s\n",b);
97                  printf("fork: %s\n", strerror(errno));
98                  result=0;
99                  }
100     return result;
101 }
```

Figura 7: Função "forkError".

A deteção da interrupção da execução (sinal normalmente relacionado com a combinação de botões *Control+C*) é feita através do envio do sinal *SIGINT* para o programa. Nesse caso, a variável global *running* irá determinar o estado de execução do programa.

No início da função “main” do programa é colocado a linha “signal(SIGINT, handler);“ que detecta o sinal *SIGINT* e invoca a função “handler” que altera o estado da variável *running* e termina o programa sem alterar o estado do ficheiro de input.

2. Funcionalidades avançadas

2.1 Acesso a resultados de comandos anteriores arbitrários

A execução dos comandos seguidos por “$N|” contém a mesma lógica de execução dos sinais $ e $| com a particularidade de que o *stdin* do novo comando vai ser o ficheiro “resultN.txt” em que N é o número entre “$” e “|”.

Para reconhecer o sinal “$N|” utiliza-se a expressão regular “ $[1-9][0-9]*| “ e a biblioteca “regex.h” é utilizada de modo a fazer-se o *parsing* da linha.

```
1  Este comando lista os ficheiros:
2  $ ls
3
4  A primeira linha do comando acima
5  $| head -1
6
7
8  Configuração da placa de rede.
9  $ ifconfig
10
11 A primeira linha do comando acima
12 $| head -1
13
14 Mostra data
15 $ date
16
17 A primeira linha do terceiro comando
18 $3| head -1
19
```

Figura 8: Ficheiro "comandos2.nb".

```
271             reti = regcomp(&regex, "$[1-9][0-9]*|", 0);
272             if( reti ){ fprintf(stderr, "Could not compile regex\n"); exit(1); }
273             /* Execute regular expression */
274             reti = regexec(&regex, line, 0, NULL, 0);
275             if( !reti && (running)){
276                 //printf("\n Match %s \n",line);
277                 //printf("DOLLAR_NUMBER\n" );
278                 char *b =  trim(line + 4);
279                 int number = atoi(&line[1]);
280                 contador++;
281                 printf("comando nº: %d \t %s\t com STDIN do comando %d\n",contador,b,number);
282                 strcatFilename(contador);
283                 // ABRIR RESULTN.TXT       Para colocar o STDOUT
284                 FILE * resultN;
285                 resultN =fopen(filename,"wr+"); //abre o resultN.txt
286                 int d=fork();
287                 if(d==0){
288                     strcatFilename(number);
289                     result1 =fopen(filename,"r");
290                     dup2(fileno(result1),0);//STDIN_FILENO
291                     dup2(fileno(resultN),1); //STDOUT_FILENO
292                     fclose(result1);
293                     fclose(resultN);
294                     execl("/bin/sh", "/bin/sh", "-c", b, NULL);
295                     exit(-1);
296                     }
297                 else{
298                     //wait(0);
299                     int status;
300                     wait(&status);
301                     flagErrorFork=forkError(status,b);
302                     if( flagErrorFork == 0){ // 0 deu erro no fork
303                         break;
304                     }
305                     fclose(result1);
306                     fputs(">>>\n",out);
307                     FILE * result2;
308                     result2 =fopen(filename,"r");
309                     //escrever para out.txt
310                     while ((read2 = getline(&line2, &len2, result2)) != -1){
311                         fputs(line2,out);
312                         }
313                     fputs("<<<\n",out);
314                     }
315             }
```

Figura 9: Parsing do comando "$n|".

2.2 Execução de conjuntos de comandos

A execução de conjuntos de comandos foi testada com o exemplo abaixo, sendo que o “execl” utilizado no programa executa o comando com pipes tal como pretendido.

```
1  Este comando lista os ficheiros:
2  $ ls -l | sort | grep "notebook"
3
4  A primeira linha do comando acima
5  $| head -1
6
```

Figura 10: Ficheiro "comandos3.nb".

Conclusão

Após duas semanas de desenvolvimento, é necesário realçar alguns pontos importantes, tal como a dificuldade de implementar funções de baixo nível. No entanto, mesmo com estes e demais obstáculos relacionados com a carência de tempo, foi possível ficar a compreender melhor as system calls, bem como o contexto no qual as usar para ter um maior controlo de processos e ficheiros.

A utilização de ficheiros temporários como *stdin* e *stdout* foi discutida pela grupo e vista como a solução para completar todos os pontos colocados no enunciado do trabalho.

A única exceção foi no ponto 2.2, no qual não se decidiu avançar para a execução em background dos ficheiros.

Como conclusão, entendemos que o trabalho efetuado se encontra adequado e completo dado que realiza os objetivos propostos no enunciado.