Mestrado Integrado em
Engenharia Informática e Computação

Sistemas Operativos
EIC0027

Relatório Final

1º Trabalho Prático: SO Shell (sosh)

Jorge Nuno Polónia Coelho Ferro, ei05037
Ricardo Daniel Pacheco Martins, ei05066

Universidade do Porto
Faculdade de Engenharia
FEUP

Faculdade de Engenharia da Universidade do Porto
Rua Dr. Roberto Frias, s/n,
4200-465 Porto, Portugal

Porto, 1 de Novembro de 2009

Resumo

Este relatório é o complemento do culminar do projecto realizado durante o semestre no âmbito da disciplina de Sistemas Operativos. Aqui serão apresentados os resultados e documentados todos os pontos necessários para a compreensão do programa desenvolvido, programa esse que visa a implementação de um interpretador de comandos simples em C.

Conteúdo

Capítulo 1

Objectivos

"Conhecer e utilizar a interface de desenvolvimento de sistemas baseados em Unix. Nomeadamente, após a conclusão deste trabalho com sucesso, os alunos serão capazes de conseguir

- criar novos processos;
- fazê-los intercomunicar por sinais;
- percorrer um sistema de ficheiros e dele obter informações." [2]

Capítulo 2

Estrutura do trabalho

O trabalho foi dividido nos seguintes módulos

- history - Módulo onde são definidas todas as funções relacionadas com a implementação do histórico;
- aux - Módulo onde são implementadas todas as funções auxiliares da shell;
- cmds - Módulos onde são implementados todos os comandos nativos da shell;
- sosh - Módulo principal do programa, onde é implementada a shell.

Capítulo 3

Funcionalidades

Foi proposta a realização de quatro versões da sosh, com as seguintes funcionalidades. Segue-se a sua listagem e implementação.

3.1 sosh 0.1

Funcionalidade	Implementada
Estar em loop a receber comandos	Sim
Reconhecer o comando *quem*	Sim
Processo pai espera por processo filho	Sim
Pergunta ao receber o sinal Ctrl-C	Sim

3.2 sosh 0.2

Funcionalidade	Implementada
Reconhecer o comando *quem*	Sim
Reconhecer o comando *psu*	Sim
Reconhecer o comando *ver*	Sim
Reconhecer o comando *ajuda*	Sim
Reconhecer o comando *localiza cmd*	Sim
Reconhecer o comando *exit*	Sim

3.3 sosh 0.3a

Funcionalidade	Implementada
Reconhecer as aplicações localizadas em /usr/bin	Sim
Organização dos comandos da sosh no módulo cmds	Sim

3.4 sosh 0.3b

Funcionalidade	Implementada
Reconhecer o comando *hist*	Sim
Reconhecer o comando *!string*	Sim

Capítulo 4

Detalhes de Implementação

4.1 Principais funções

Para os membros do grupo, as principais funções deste trabalho, tanto pelo seu nivel de complexidade como pela qualidade da solução encontrada são, por ordem

1. *Comando localiza cmd*
2. *Comando hist*
3. *Comando !string*
4. *Aplicações localizadas em /usr/bin*

De modo a implementá-las, foram usadas as seguintes abordagens.

4.1.1 Comando localiza cmd

Ao chamar a função na sosh, é chamada a função *cmd_localiza* definido em *cmds.h*, tendo sido alterada a sua definição para incluir como parametro cmd, que se trata de toda a linha introduzida pelo utilizador.
Uma vez que este trabalho é académico, é incluido na função *cmd_localiza* um path inicial de modo a tornar testes e demonstrações mais simples.
Esta função *cmd_localiza* retira do parametro cmd os caracteres "localiza "chamando de seguida a função *depthsearch* que, como o próprio nome indica, realiza uma pesquisa em profundidade na árvore de ficheiros. Quando encontra um directório que inclui a *string* desejada pelo utilizador chama a função auxiliar *print_tree_below* que imprime todo o ramo da árvore de ficheiros abaixo desse directório, uma vez que nesse ramo, todos os ficheiros

vão conter a *string* pesquisada pelo utilizador no seu *path* absoluto.
Se encontra um directório que não possui a *string* desejada pelo utilizador, faz um *fork* que realiza uma chamada recursiva da função para esse directório continuando assim a sua busca em profundidade.
É tido o cuidado de verificar se os directórios não são *symlinks*, o que faria o programa entrar em ciclo (código fonte na página 13).

4.1.2 Comando hist

O histórico é implementado recorrendo a uma lista duplamente ligada implementada no módulo *history* inicializada num *scope* global. De cada vez que um comando é introduzido, é acrescentado ao final dessa lista sendo essa mesma lista percorrida através dos apontadores *next* e *previous* presentes em cada nó (código fonte na página 16).

4.1.3 Comando !string

O programa foi construido de modo a ter uma função (*soshreadline*) que recebe o input do utilizador, trata-o e devolve ao corpo principal da aplicação a "*string* limpa"que despoleta a chamada da função. No interior dessa função, se o input começar por um ponto de exclamação, é feita uma pesquisa no histórico por um comando previamente introduzido que possua a *string* fornecida pelo utilizador. Caso encontre, passa esse comando ao corpo principal da aplicação para despoletar a chamada da função correspondente, adicionando esse comando ao histórico e não a pesquisa introduzida pelo utilizador (código fonte na página 18).

4.1.4 Aplicações localizadas em /usr/bin

Esta função foi realizada utilizando uma implementação de uma função denominada *makeargv* presente no livro Unix Systems Programming[1], que recebe uma *string* dividindo-a em *tokens* construindo um *array* de *strings* que é de seguida passado à função *execv*.

Capítulo 5

Conclusões

Após um período de investigação antes da fase de programação, foi marcada como função com grau de dificuldade mais elevado a função *localiza cmd*.
Essa função exigiu um planeamento mais detalhado, uma vez que com as mudanças de *path* e com os sucessivos *forks*, o output tornava-se bastante complexo, o que fazia com que fosse necessário ter sempre a noção de cada passo da função. Estas dificuldades foram debeladas introduzindo no código produzido *error handling* em todas as chamadas ao sistema tornando-o por isso muito verboso no caso de eventuais erros.
Foram utilizadas as bibliotecas *GNU readline* e *GNU history* numa fase inicial do projecto, tendo sido posteriormente abandonadas após uma conversa com o docente, uma vez que esse impedimento não se encontrava discriminado no enunciado. Esse factor alterou a marcação do grau de dificuldade das funções *hist* e *!string* no plano de desenvolvimento e obrigou também a uma reestruturação do código do programa.
No final, o grupo considera que foram concluídas com sucesso todas as etapas deste trabalho devendo-se isso a um planeamento e investigação cuidadosos em vez de programação errática. Apesar de todas as etapas terem sido concluídas, todos os objectivos elaborados nesse planeamento não foram cumpridos, havendo por isso possibilidade de melhoramentos numa versão futura(ver cap. 6).

Capítulo 6

Melhoramentos

No final deste desenvolvimento, usando o comando

```
grep -R TODO *
```

verificam-se duas situações marcadas como necessitando de melhoramento durante o desenvolvimento.
A primeira delas prende-se com a pergunta ao utilizador aquando da recepção do ctrl-c. O planeamento exigia que a função fosse *async signal safe* usando para isso apenas funções que o fossem como o write e o read e evitando funções que não o fossem como o strcmp. Esse objectivo não foi cumprido, não havendo um controlo total sobre o input do utilizador controlando o programa apenas inputs de uma letra.
A segunda situação prende-se com o sinal enviado para a finalização do programa.
Além disto, o grupo considera a utilização das bibliotecas GNU readline e GNU history um melhoramento ao programa uma vez que expandiria em muito as funcionalidades da sosh. Podem ser verificados estes melhoramentos na branch *master* do sistema de controlo de versões utilizado pelo grupo(ver cap. 7) estando implementadas todas as funcionalidades do programa excepto a função *localiza cmd*.

Capítulo 7

Ambiente de desenvolvimento

Este projecto foi desenvolvido usando os seguintes recursos:

- Sistema Opertativo: GNU/Linux
- Linguagem: C
- Sistema de controlo de versões: Git (git://github.com/nunopolonia/sope_tp1.git)
- Sistema tipográfico para preparação de documentos: LaTeX
- Ver Bibliografia (pág. 11)

Bibliografia

[1] Steven Robbins and Kay A. Robbins. *Unix system programming: communication, concurrence, and threads*. Prentice Hall Professional Technical Reference, Upper Saddle River, New Jersey 07458, 1995.

[2] Hugo Ferreira Rui Maranhão, José Vila Verde. 1º trabalho prático: So shell (sosh). http://web.fe.up.pt/ rma/SOPE/TP1.pdf, Outubro 2009.

Capítulo 8

Anexos

8.1 Declaração de autoria

Declaramos sob compromisso de honra que este trabalho, nas suas partes constituintes de código e relatório, é original e da nossa autoria, não correspondendo, portanto, a cópia ou tradução de outros trabalhos já realizados, na FEUP ou fora dela.
Mais declaramos que todos os documentos ou código que serviram de base ao desenvolvimento do trabalho descrito no relatório e seus anexos são adequadamente citados e explicitados na respectiva secção de referências bibliográficas e que todas as eventuais partes transcritas ou utilizadas de outras fontes estão devidamente assinaladas, identificadas e evidenciadas.

Os autores,
Jorge Nuno Polónia Coelho Ferro
Ricardo Daniel Pacheco Martins

8.2 Comando localiza cmd

```c
/* cmds.c */

int cmd_localiza(char* cmd) {
  char *init_path = "/home/nunopolonia/work/inqueritos";
  char search_string[MAX_CANON];

  /* Returns search string without "localiza " */
  strcpy(search_string, cmd+9);

  /* makes a depth-first search starting at "init_path" searching for search_string */
  depthsearch(init_path, search_string);

  return 0;
}

/* aux.c */

void depthsearch(char *path, char *search_string) {
  struct dirent *direntp;
  struct stat statbuf;
  char mycwd[PATH_MAX];
  DIR *dirp;
  pid_t childpid, waitchild;

  if( chdir(path) == -1 )
    perror("Failed to change working directory");

  /* path fetching error handling */
  if( getcwd(mycwd, PATH_MAX) == NULL ) {
    perror("Failed to get current working directory");
    return;
  }

  /* open the init_path directory */
  if( (dirp = opendir(mycwd) ) == NULL ) {
    perror("Failed to open directory");
    return;
  }

  while( ( direntp = readdir(dirp) ) != NULL ) {
    /* File status error handling */
    if(stat(direntp->d_name, &statbuf) == -1 )
      perror("Failed to get file status");

    /* If the string is in any of the folder files */
    if(strstr(direntp->d_name, search_string) != NULL ) {
      /* if its a dir print all the tree below in a new process so
      ** we don't mess with the current working directory */
      if( (S_ISDIR(statbuf.st_mode) == TRUE) && (S_ISLNK(statbuf.st_mode) == FALSE) ) {
        printf("%s/%s\n", mycwd, direntp->d_name);

        childpid = fork();

        /* fork error handling */
        if( childpid == -1 ) {
          perror("Failed to fork\n");
          return;
        }
        /* child code */
```

```
60          if( childpid == 0 ) {
61            print_tree_below(direntp->d_name);
62            return;
63          }
64          /* parent process waiting for children */
65          while( ( waitchild = waitpid(-1, NULL, WNOHANG) ) ) {
66            if( (waitchild == -1) && (errno != EINTR) )
67              break;
68          }
69          continue;
70        /* if its a file print it */
71        } else
72          printf("%s/%s\n", mycwd, direntp->d_name);
73          continue;
74      /* else if its a directory fork and run this again */
75      } else if( (S_ISDIR(statbuf.st_mode) == TRUE) && (S_ISLNK(statbuf.st_mode) == FALSE)
76                 && (strcmp(direntp->d_name, ".") != 0) && (strcmp(direntp->d_name, "..") != 0) )
77        /* Launches a child process for each folder */
78        childpid = fork();
79
80        /* fork error handling */
81        if( childpid == -1 ) {
82          perror("Failed to fork\n");
83          return;
84        }
85        /* child code */
86        if( childpid == 0 ) {
87          depthsearch(direntp->d_name, search_string);
88          return;
89        }
90
91        /* parent process waiting for children */
92        while( ( waitchild = waitpid(-1, NULL, WNOHANG) ) ) {
93          if( (waitchild == -1) && (errno != EINTR) )
94            break;
95        }
96      }
97    }
98    /* close init_path directory */
99    while( (closedir(dirp) == -1) && (errno == EINTR) );
100
101   return;
102 }
103
104 void print_tree_below(char *path) {
105   struct dirent *direntp;
106   struct stat statbuf;
107   char mycwd[PATH_MAX];
108   DIR *dirp;
109   pid_t childpid, waitchild;
110
111   if( chdir(path) == -1 )
112     perror("Failed to change working directory");
113
114   /* path fetching error handling */
115   if( getcwd(mycwd, PATH_MAX) == NULL ) {
116     perror("Failed to get current working directory");
117     return;
118   }
119
120   /* open the init_path directory */
121   if( ( dirp = opendir(mycwd) ) == NULL ) {
```

```
122       perror("Failed to open directory");
123       return;
124     }
125
126     while( ( direntp = readdir(dirp) ) != NULL ) {
127       /* File status error handling */
128       if(stat(direntp->d_name, &statbuf) == -1 )
129         perror("Failed to get file status");
130
131       /* Print the filenames */
132       if( (strcmp(direntp->d_name, ".") != 0) && (strcmp(direntp->d_name, "..") != 0) )
133         printf("%s/%s\n", mycwd, direntp->d_name);
134
135       if( (S_ISDIR(statbuf.st_mode) == TRUE) && (S_ISLNK(statbuf.st_mode) == FALSE)
136           && (strcmp(direntp->d_name, ".") != 0) && (strcmp(direntp->d_name, "..") != 0) ) {
137         /* Launches a child process for each folder */
138         childpid = fork();
139
140         /* fork error handling */
141         if( childpid == -1 ) {
142           perror("Failed to fork\n");
143           return;
144         }
145         /* child code */
146         if( childpid == 0 ) {
147           print_tree_below(direntp->d_name);
148           return;
149         }
150
151         /* parent process waiting for children */
152         while( ( waitchild = waitpid(-1, NULL, WNOHANG) ) ) {
153           if( (waitchild == -1) && (errno != EINTR) )
154             break;
155         }
156       }
157     }
158     /* close init_path directory */
159     while( (closedir(dirp) == -1) && (errno == EINTR) );
160
161     return;
162   }
```

8.3 Comando hist

```
1  /* history.h */
2
3  /* declaration of global scoped list of commands entered */
4  history_t *history_list;
5
6  /* history.c */
7  void using_history() {
8     history_list = (history_t*) malloc(sizeof(history_t));
9     history_list->first = NULL;
10    history_list->last = NULL;
11
12    return;
13 }
14
15 void history_destroy(history_t *history) {
16    free(history);
17
18    return;
19 }
20
21
22 void history_add(char *string) {
23    history_item_t *new = NULL, *previous_last = NULL;
24    char *new_string;
25
26    /* since we're using a single memory block for every command, we have to create space in
27    ** memory for each command in history or else they would all point to the last command */
28    new = (history_item_t*) malloc(sizeof(history_item_t));
29    new_string = malloc(sizeof(char)*(strlen(string)+1));
30    strcpy(new_string, string);
31
32    /* initialization of the new item structure */
33    new->string = new_string;
34    new->next = NULL;
35    new->previous = NULL;
36
37    /* if the list is not empty */
38    if(history_list->last != NULL) {
39     previous_last = history_list->last;
40     previous_last->next = new;
41     new->previous = previous_last;
42     history_list->last = new;
43    /* if the command is the first inserted */
44    } else {
45     history_list->first = new;
46     history_list->last = new;
47    }
48
49   return;
50 }
51
52 void history_print() {
53   history_item_t *current = history_list->first;
54   int count = 1;
55
56   while (current != NULL) {
57      printf("%d: %s\n", count, current->string);
58      current = current->next;
59      count++;
```

```
60    }
61
62    return;
63  }
64
65  /* cmds.c */
66  int cmd_hist() {
67
68    history_print(history_list);
69
70    return 0;
71  }
72
73  /* aux.c */
74
75  /* sosh readline function */
76  void soshreadline(char *clean_line) {
77      char line[MAX_CANON] = "";
78      int line_size = 0;
79      char *command = NULL;
80
81      /* cleans the input buffer */
82    memset(clean_line, 0, MAX_CANON);
83
84      printf("> ");
85      if( fgets(line, MAX_CANON, stdin) != NULL ) {
86      line_size = strlen(line);
87      strncpy(clean_line, line, line_size-1);
88
89      /* adds the new command to history except if it's a search */
90      if(strncmp(clean_line, "!", 1) != 0)
91        history_add(clean_line);
92      else {
93        /* if it's a search, finds the last command sends it to processing
94        ** instead of the line received in the stdin */
95        command = history_search(clean_line);
96
97        /* if a command was found that satisfies the search */
98        if(command != NULL) {
99          history_add(command);
100         printf("command found: %s\n", command);
101         /* clears the memory block used to received every command,
102         ** we pass that command for parsing */
103         memset(clean_line, 0, MAX_CANON);
104         line_size = strlen(command);
105         strncpy(clean_line, command, line_size);
106       } else
107         printf("Command not found in history\n");
108     }
109   }
110
111     return;
112 }
```

8.4 Comando !string

```c
/* history.h */

/* declaration of global scoped list of commands entered */
history_t *history_list;

/* history.c */
char *history_search(char *cmd) {
  history_item_t *current = history_list->last;
  char *search_string = NULL;

  /* remove the ! from the beginning of the sentence */
  search_string = strtok(cmd, "!");

  /* search the history backwards */
  while (current != NULL) {
     if( strstr(current->string, search_string) != NULL )
        return current->string;
     else
      current = current->previous;
  }

  return NULL;
}

/* aux.c */

/* sosh readline function */
void soshreadline(char *clean_line) {
    char line[MAX_CANON] = "";
    int line_size = 0;
    char *command = NULL;

    /* cleans the input buffer */
  memset(clean_line, 0, MAX_CANON);

    printf("> ");
    if( fgets(line, MAX_CANON, stdin) != NULL ) {
    line_size = strlen(line);
    strncpy(clean_line, line, line_size-1);

    /* adds the new command to history except if it's a search */
    if(strncmp(clean_line, "!", 1) != 0)
      history_add(clean_line);
    else {
      /* if it's a search, finds the last command sends it to processing
      ** instead of the line received in the stdin */
      command = history_search(clean_line);

      /* if a command was found that satisfies the search */
      if(command != NULL) {
        history_add(command);
        printf("command found: %s\n", command);
        /* clears the memory block used to received every command,
        ** we pass that command for parsing */
        memset(clean_line, 0, MAX_CANON);
        line_size = strlen(command);
        strncpy(clean_line, command, line_size);
      } else
        printf("Command not found in history\n");
```

```
60          }
61      }
62
63          return;
64  }
```