Shellproject.pdf

GeXx_

Sistemas Operativos

2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingeniería Informática
Universidad de Málaga

```
/**
UNIX Shell Project

Sistemas Operativos
Grados I. Informatica, Computadores & Software
Dept. Arquitectura de Computadores - UMA

Some code adapted from "Fundamentos de Sistemas Operativos", Silberschatz
et al.

To compile and run the program:
   $ gcc Shell_project.c job_control.c -o Shell
   $ ./Shell
      (then type ^D to exit program)


-Mask m√∫ltiple
-Time-out
-Pipe simple
-Historial

**/

#include "job_control.h"   // remember to compile with module
job_control.c
#include <string.h>
#include <sys/mman.h>
#include <errno.h>
#define MAX_LINE 256 /* 256 chars per line, per command, should be
enough. */

//-------------------------PARA HISTORIAL-------------------------------
-----//
typedef struct cmm{
      char * command;
      int id;
      struct cmm * prev;
      struct cmm * next;

} command;
//-----------------------------------------------------------------------
----//

job * background_list;

//Colors
#define NORMAL "\033[0m"
#define AMARILLO "\x1b[93m"
#define ROJO "\x1b[31;1;1m"
#define NEGRO "\x1b[0m"
#define VERDE "\x1b[32;1;1m"
#define AZUL "\x1b[34;1;1m"
#define CIAN "\x1b[36;1;1m"
#define MARRON "\x1b[33;1;1m"
#define PURPURA "\x1b[35;1;1m"

// ---------------------------------------------------------------------
-
```

```
//HANDLER
// ------------------------------------------------------------------------
-

void handler(int signal){

      int pid_wait;
      int status;
      int info;
      job * jobs = background_list;
      enum status status_res;

      block_SIGCHLD();

      while(jobs!=NULL){
            pid_wait = waitpid(jobs->pgid, &status, WUNTRACED | WNOHANG |
WCONTINUED);
            status_res = analyze_status(status,&info);
            if(pid_wait == jobs->pgid){
                  printf(AZUL"%s pid: %d, command: %s, %s, info: %d%s\n",
                  state_strings[BACKGROUND], jobs->pgid, jobs->command,
status_strings[status_res],info,NEGRO);
                  if(status_res == SUSPENDED){
                        jobs->state = STOPPED;
                        jobs = jobs->next;
                  }else if(status_res == CONTINUED){
                        jobs->state = BACKGROUND;
                        jobs = jobs->next;
                  }else{
                        job * aux= jobs->next;
                        delete_job(background_list,jobs);
                        jobs=aux;
                  }
            } else {
                  jobs = jobs->next;
            }
      }
      unblock_SIGCHLD();
}

//FUNCIONES √öTILES

void printCommand(char *cmm[]) {
      int i = 0;
      while (cmm[i] != NULL) {
            printf("%s ", cmm[i]);
            i++;
      }
      printf("\n");
}

char getch() {
      int shell_terminal = STDIN_FILENO;
      struct termios conf;
      struct termios conf_new;
      char c;
      tcgetattr(shell_terminal,&conf); /* leemos la configuracion actual
*/
```

```
      conf_new = conf;
      conf_new.c_lflag &= (~(ICANON|ECHO));
      conf_new.c_cc[VTIME] = 0;
      conf_new.c_cc[VMIN] = 1;
      /* configuramos sin buffer ni eco */
      tcsetattr(shell_terminal,TCSANOW,&conf_new); /* establecer
configuracion */
      c = getc(stdin); /* leemos el caracter */
      tcsetattr(shell_terminal,TCSANOW,&conf); /* restauramos la
configuracion */
      return c;
}

void resetLine(char cmd[]) {
      int i = 0, max = strlen(cmd);
      for (i; i < max; i++) printf(" ");
}

void readInput(command *history, char inputBuffer[]) {
      /* Las teclas de cursor devuelven una secuencia de 3 caracteres, 27
- 91 -
      (65, 66, 67 √≥ 68) */
      printf("\033[s");
      command *aux = NULL;
      if (history != NULL) {
            aux = (command *) malloc(sizeof(struct cmm));
            aux->command = NULL;
            aux->next = history->next;
            aux->prev = history;
      }

      command *pointer = aux;

      char trasCursor[MAX_LINE+1];
      trasCursor[0] = '\0';
      int tamTrasCursor = 0;

      int readCmd = 0, idBuff = 0, tamInput = 0, cont = 1;
      char sec[3];

      int i, j;
      char c;


      while (cont) {
      sec[0] = getch();
      switch (sec[0]){
            case 27:
                  sec[1] = getch();
                  if (sec[1] == 91) // 27,91,...
                  {
                        sec[2] = getch();
                        switch (sec[2]){
                              case 65: /* ARRIBA */
                                    if (history != NULL && pointer !=
history->next) {
                                          printf("\033[u");
                                          resetLine(inputBuffer);
```

```
                                        printf("\033[u");
                                        pointer = pointer->prev;
                                        strcpy(inputBuffer, pointer-
>command);
                                        idBuff = strlen(pointer-
>command);
                                        printf("%s", pointer->command);
                                        trasCursor[0] = '\0';
                                        tamTrasCursor = 0;
                                  }
                                  break;
                                  case 66: /* ABAJO */
                                        if (history != NULL) {
                                              if (pointer->next ==
history->next) pointer = aux;
                                              else pointer = pointer-
>next;

                                              printf("\033[u");
                                              resetLine(inputBuffer);
                                              printf("\033[u");
                                              idBuff = 0;
                                              if (pointer != aux) {
                                                    strcpy(inputBuffer,
pointer->command);
                                                    idBuff =
strlen(pointer->command);
                                                    printf("%s", pointer-
>command);
                                              }
                                              trasCursor[0] = '\0';
                                              tamTrasCursor = 0;
                                        }
                                        break;
                                  case 67: /* DERECHA */
                                        if (tamTrasCursor != 0) {
                                              printf("\033[1C");
                                              idBuff++;
                                              i = 0;
                                              for (i; i < tamTrasCursor;
i++) {
                                                    trasCursor[i] =
trasCursor[i+1];
                                              }
                                              tamTrasCursor--;
                                        }
                                        break;
                                  case 68: /* IZQUIERDA */
                                        if (idBuff != 0) {
                                              printf("\033[1D");
                                              idBuff--;
                                              c = inputBuffer[idBuff];
                                              i = tamTrasCursor;
                                              for (i; i >= 0; i--) {
                                                    trasCursor[i+1] =
trasCursor[i];
                                              }
                                              tamTrasCursor++;
```

```
                                        trasCursor[0] = c;
                                    }
                                    break;
                              }
                        }
                        break;
                  case 127: /* BORRAR */
                        if (idBuff > 0) {
                              printf("\033[1D%s \033[1D", trasCursor);
                              i = tamTrasCursor;
                              for (i; i > 0; i--) {
                                    printf("\033[1D");
                              }
                              i = idBuff;
                              for (i; i < tamInput; i++) {
                                    inputBuffer[i-1] = inputBuffer[i];
                              }
                              idBuff--;
                              tamInput--;
                              i = 0;
                              j = idBuff;
                              while (i < tamTrasCursor) {
                                    inputBuffer[j] = trasCursor[i];
                                    i++;
                                    j++;
                              }
                        }
                        break;
                  case 4:                 /* ^D */
                        inputBuffer[0] = sec[0];
                        cont = 0;
                        break;
                  case 10:          /* \n */
                        cont = 0;
                        i = idBuff;
                        j = 0;
                        for (i; i < idBuff+tamTrasCursor; i++) {
                              inputBuffer[i] = trasCursor[j];
                              j++;
                        }
                        idBuff = i;
                        inputBuffer[idBuff] = '\n';
                        inputBuffer[idBuff+1] = '\0';
                        printf("\n");
                        break;
                  default:          /* CUALQUIER OTRO CARACTER */
                        tamInput++;
                        printf("%c%s", sec[0], trasCursor);
                        i = tamTrasCursor;
                        for (i; i > 0; i--) {
                              printf("\033[1D");
                        }
                        inputBuffer[idBuff] = sec[0];
                        idBuff++;
                  }
            }
            free(aux);
}
```

```
void addCommand(command *aux, command **history, int *tamHistory) {
      if (*history == NULL) {
            *history = aux;
            (*history)->id = 1;
            (*history)->prev = *history;
            (*history)->next = *history;
            (*tamHistory) = 1;
      } else {
            aux->prev = *history;
            aux->next = (*history)->next;
            aux->id = ((*history)->id) + 1;
            ((*history)->next)->prev = aux;
            (*history)->next = aux;
            (*history) = aux;
            (*tamHistory) = (*tamHistory) + 1;
      }
}


int main(void)
{
      char inputBuffer[MAX_LINE]; /* buffer to hold the command entered
*/
      int background;             /* equals 1 if a command is followed by
'&' */
      char *args[MAX_LINE/2];     /* command line (of 256) has max of 128
arguments */
      // probably useful variables:
      int pid_fork, pid_wait; /* pid for created and waited process */
      int status;             /* status returned by wait */
      enum status status_res; /* status processed by analyze_status() */
      int info;                   /* info processed by analyze_status()
*/

      background_list = new_list("JOBS LIST");
      command *history = NULL;    /* Apunta al √ltimo elemento de la
lista, siendo esta circular */
      int tamHistory;
      char * comando [MAX_LINE/2]; /* array auxiliar */
      char * comando2 [MAX_LINE/2]; /* otro array auxiliar */

      //-----------PARA MASK-------------//

      int esMascara=0;
      int mascaras[MAX_LINE/2];

      //-----------PARA TIMEOUT-------------//

      int timeout, time, pid_timeout;

      ignore_terminal_signals();
      signal(SIGCHLD,handler);

      while (1)   /* Program terminates normally inside get_command()
after ^D is typed*/
      {
            int pipes = 0;
```

```
            int cont = 0;
            comando[0] = NULL;
            comando2[0] = NULL;
            command *aux;
            timeout = 0;
            char dir[200];
            printf(AZUL"[%s] ", getcwd(dir,199),NEGRO);
            printf(VERDE"COMMAND->"NEGRO);
            fflush(stdout);
            readInput(history, inputBuffer);
            aux = (command *) malloc(sizeof(struct cmm));
            aux->command = strdup(inputBuffer);
            aux->command[strlen(aux->command)-1] = '\0';
            addCommand(aux, &history, &tamHistory);

            com:
            get_command(inputBuffer, MAX_LINE, args, &background);  /* get
next command */

            if(args[0]==NULL) continue;   // if empty command

            //Buscamos si hay pipes y dejamos los comandos listos
            for(int i=0; args[i] != NULL; ++i) {
                  if(!strcmp(args[i],"|")) {
                        pipes = 1;
                        comando[i] = NULL;
                  }
                  else if(!pipes) {
                        comando[i] = strdup(args[i]);
                  }
                  else {
                        comando2[cont] = strdup(args[i]);
                        if(args[i+1] == NULL) comando2[cont+1] = NULL;
                        ++cont;
                  }
            }

            if(pipes) {
            //-----------PIPE-------------//
                  int pid_continue = fork();


                  if(!pid_continue) {
                        if(comando[0] != NULL && comando2[0] != NULL) {
                              int desc[2];
                              int fno;

                              pipe(desc);
                              if(fork())
                              {// proceso padre ->
                                    fno = fileno(stdout);
                                    dup2(desc[1],fno);
                                    close(desc[0]);
                                    execvp(comando[0], comando);
                              }else
                              { // proceso hijo ->
                                    fno = fileno(stdin);
                                    dup2(desc[0],fno);
```

```
                                    close(desc[1]);
                                    execvp(comando2[0], comando2);
                              }
                        }else{
                              puts("error: wrong commands");
                              continue;
                        }
                  }
                  sleep(100);
                  continue;
            }else if (strcmp(args[0],"cd") == 0) {
            //-----------CD-------------//
                  if (args[1] == NULL) chdir(getenv("HOME"));
                  else chdir(args[1]);
                  continue;
            } else if (strcmp(args[0],"jobs") == 0) {
            //----------JOBS------------//
                  if (empty_list(background_list)) printf(PURPURA"No jobs
to show\n"NEGRO);
                  else {
                        print_job_list(background_list);
                        printf("\n");
                  }
                  continue;
            } else if (strcmp(args[0],"fg") == 0) {
                  //-----------FG-------------//
                  job * aux;
                  if (args[1] == NULL) aux =
get_item_bypos(background_list, 1);
                  else aux = get_item_bypos(background_list,
atoi(args[1]));

                  if (aux == NULL) printf(ROJO"Empty jobs list\n"NEGRO);
                  else {
                        aux->state = FOREGROUND;
                        printf(AZUL"%s pid: %d, command: %s%s\n",
                        state_strings[aux->state], aux->pgid, aux-
>command, NEGRO);

                        set_terminal(aux->pgid);
                        killpg(aux->pgid, SIGCONT);
                        pid_wait = waitpid(aux->pgid, &status, WUNTRACED);
                        set_terminal(getpid());
                        status_res = analyze_status(status, &info);

                        if (status_res == SUSPENDED) {
                              aux->state = STOPPED;
                        } else delete_job(background_list, aux);

                        printf(AZUL"%s pid: %d, command: %s, %s, info:
%d%s\n",
                        state_strings[FOREGROUND], pid_fork, args[0],
                        status_strings[status_res], info, NEGRO);
                  }
                  continue;
            } else if (strcmp(args[0],"bg") == 0) {
            //-----------BG-------------//
                  job* auxiliar;
```

```
            int id;

            if(args[1] != NULL) {
                id = atoi(args[1]);

            }else{
                id = 1;
            }

                 block_SIGCHLD();
            auxiliar = get_item_bypos(background_list, id);

            if(auxiliar == NULL) {
                printf(ROJO"Lista vac√≠a\n"NEGRO);

            }else{
                auxiliar->state = BACKGROUND;       // Cambiamos estado a
BACKGROUND

                killpg(auxiliar->pgid, SIGCONT);    // Se√±al para que el
grupo siga

                printf(MARRON"%s pid: %d, command: %s%s\n",
                            state_strings[auxiliar->state], auxiliar-
>pgid, auxiliar->command, NEGRO);

            }
                 unblock_SIGCHLD();
             continue;
            } else if (strcmp(args[0], "historial") == 0) {
            //--------HISTORIAL---------//
                 rep:
                 aux = history->next;
                 if (args[1] == NULL) {
                       do {
                             printf("%d %s\n", aux->id, aux->command);
                             aux = aux->next;
                       } while (aux->prev != history);
                       continue;
                 } else {
                       int i = atoi(args[1]);
                       if (tamHistory == 1) {
                             printf(ROJO"Historial vacio%s\n", NEGRO);
                       } else if (i < 1 || i > tamHistory) {
                             printf(ROJO"Deber ser un argumento y
positivo%s\n", NEGRO);
                       } else {
                             for (i; i > 1; i--) {
                             aux = aux->next;
                             }
                             aux->command[strlen(aux->command)] = '\n';
                             strcpy(inputBuffer, aux->command);
                             aux->command[strlen(aux->command)-1] = '\0';
                             if (strcmp(args[0], "historial") == 0) goto
rep;
                             goto com;
                       }
                 }
```

```
            } else if (strcmp(args[0], "time-out") == 0) {
            //---------TIME-OUT---------//
                  if (args[1] != NULL && args[2] != NULL) {
                        time = atoi(args[1]);
                        if(time <=0){
                              printf(ROJO"El tiempo debe ser un natural
positivo...¬øTe gusta jugar a ser Dios?\n", NEGRO);
                              continue;
                        }
                        timeout = 1;
                        int i = 0;
                        while (args[i] != NULL) {
                              args[i] = args[i+2];
                              i++;
                        }
                        args[i-2] = NULL;
                        pid_fork = fork();
                  } else {
                        printf(ROJO"Faltan argumentos...\n", NEGRO);
                        continue;
                  }
            } else if(strcmp(args[0],"mask")==0){
            //---------MASK---------//
                  esMascara = 1;
                  int hayc=1;
                  int senNeg=0;
                  int i=0;

                  if(args[i+1]==NULL || strcmp(args[i+1],"-c")==0){
                        printf(ROJO"¬°¬°Donde est√° esa se√±al que yo la
vea!!\n",NEGRO);
                        continue;
                  } else {

                        //buscamos si hay '-c'
                        while(args[i+1]!=NULL && strcmp(args[i+1],"-
c")!=0){
                              i++;
                        }
                        if(args[i+1]==NULL){ //se ha parado en un -c
                              hayc=0;
                        }

                        if(hayc==0){
                              printf(ROJO"Ponle -c, hombre...\n",NEGRO);
                              continue;
                        }else{
                              if(args[i+2]==NULL){
                                    printf(ROJO"Despues de -c ponle el
comando...\n",NEGRO);
                                    continue;
                              }else {
                                    //guardamos las se√±ales y miramos si
hay alguna negativa
                                    i=0;
                                    while(strcmp(args[i+1],"-c")!=0){
                                          if(atoi(args[i+1])<=0){
                                                senNeg=1;
```

```
                                        }
                                        mascaras[i]=atoi(args[i+1]);
                                        i++;
                                  }
                                  if(senNeg){
                                        printf(ROJO"Se√±ales negativas o
0... lo que me faltaba por ver\n",NEGRO);
                                        continue;
                                  }else{
                                        mascaras[i]= NULL; //ponemos el
ultimo a NULL para poder recorrerla luego

                                        //guardar bien los comandos
                                        int m=i+2; //el siguiente de -c
                                        int k=0;
                                        while(args[m]!=NULL){ //
reescribimos el comando quitando mask, se√±ales y -c
                                              args[k]=args[m];
                                              args[m]=NULL;
                                              k++;
                                              m++;
                                        }
                                        args[k]=NULL;
                                        pid_fork = fork();
                                  }
                            }
                      }
                }
          }else { pid_fork = fork(); }

          //EXTERNO

          if(pid_fork==0){ //Hijo
                      pid_fork = getpid();
                      new_process_group(pid_fork);
                      restore_terminal_signals();

                      //PARA MASCARA
                      if(esMascara){
                            int cont=0;
                            while(mascaras[cont]!=NULL){
                                  block_signal(mascaras[cont],1);
                                  cont++;
                            }
                      }

                      execvp(args[0],args);

                      printf(ROJO"Error, command not found: %s\n",
args[0],NEGRO);
                      exit(errno);
                      break;
          }else{
                      if(timeout){
                            pid_timeout=fork();
                            if (pid_timeout == 0) {
                                  sleep(time);
                                  killpg(pid_fork, SIGKILL);
```

```
                                    exit(0);
                              }
                        }
                        if(!background){ //FOREGROUND
                              set_terminal(pid_fork); //terminal al hijo
                              //Para tener en cuenta la suspensi√≥n del
hijo -> WUNTRACED
                              pid_wait=waitpid(pid_fork,&status,WUNTRACED);
                              //devolvemos terminal al shell
                              set_terminal(getpid());
                              status_res = analyze_status(status, &info);

                               if(status_res == SUSPENDED) {   // Si se
suspende, debe almacenarse STOPPED.
                                    block_SIGCHLD();
                                    add_job(background_list,
new_job(pid_fork, args[0], STOPPED));
                                    unblock_SIGCHLD();
                              }
                              printf(AZUL"%s pid: %d, command: %s, %s,
info: %d%s\n",
                              state_strings[FOREGROUND], pid_fork, args[0],
                              status_strings[status_res], info, NEGRO);
                              fflush(stdout);
                        }else{ //BACKGROUND
                              block_SIGCHLD();
                              add_job(background_list, new_job(pid_fork,
args[0], BACKGROUND));
                              unblock_SIGCHLD();

                              printf(MARRON"%s job running... pid: %d,
command: %s%s\n",
                              state_strings[BACKGROUND], pid_fork, args[0],
NEGRO);
                        }
            }
            //vuelve a get_command
      } // end while
}
```