Practica obligatoria: MiniShell

Andrés Tena de Tena, Eric Mellado Acevedo

Sistemas Operativos - Grado en Ingeniería de Computadores Curso 2022-23

13/05/2023

Indice

Resumen

En esta práctica se estudiará el uso, implementación y funcionamiento de la creación de diferentes procesos hijos, los cuales se encargarán de las diferentes funciones con las que trabaja habitualmente una shell.

Se estudiará también cómo se han implementado las diferentes funciones como puede ser la función “Jobs”, la función “cd” y demás de como se ha llevado a cabo el paso de procesos a background y lectura de líneas en consola.

Palabras clave: Jobs, cd, background, consola, shell.

Autores

Los autores de esta práctica han sido Andrés Tena de Tena y Eric Mellado Acevedo pertenecientes al Grado de Ingeniería de Computadores.

Descripción del código

2.1 Librerias

#include "parser.h" //Librería específica proporcionada por el profesor

Librerías específicas para el uso de diferentes funciones de C.

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <ctype.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include <stdbool.h>

#include <sys/wait.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <signal.h>

Variables

```
char buf[1024];
tline * line;
int i;
pid_t pid;
int fdinput, fdoutput;
int status;
procesos = (struct proceso *)malloc(num_proceso * sizeof(struct proceso));
```

Estas variables serán las principales que se utilizarán para usar más adelante, por ejemplo, al crear procesos con pid_t, los file descriptor con fdinput y fdoutput, tline se utilizara para que junto con tokenize podamos leer las líneas que escribamos en la consola y podamos utilizar la librería proporcionada.

Además, se crea un array dinámico de procesos en el cual se añadirá todo aquel proceso que pase a estar en segundo plano.

Funciones

```
int unicoMandato(tline * line, int num_proceso, int status){
        char buffDirectorio[512];
        char *simb;
        if(!(strcmp(line->commands[0].argv[0], "cd"))){
                if(line->commands[0].argc < 2){
                        if(line->commands[0].argc == 1){
                                chdir(getenv("HOME"));
```

```
                                        if(getcwd(buffDirectorio, sizeof(buffDirectorio)) !=
NULL){
                                                printf("El directorio actual es: %s",
buffDirectorio);
                                        }else{
                                        printf("Error al obtener el directorio actual.");
                                        }
                                return 0;
                        }
                        fprintf(stderr, "Error: no se especificó un directorio como
segundo argumento al que cambiar.");
                        return 1;
                }else{
                        if(line->commands[0].argc == 2){
                                if(getcwd(buffDirectorio, sizeof(buffDirectorio)) ==
NULL){
                                        perror("Error al cambiar de directorio.");
                                        return 0;
                                }else{
                                        chdir(line->commands[0].argv[1]);
                                        if(getcwd(buffDirectorio, sizeof(buffDirectorio)) !=
NULL){
                                                printf("El directorio actual es: %s",
buffDirectorio);
                                        }else{
```

```
                                        printf("Error al obtener el directorio actual.");
                                        }
                                }
                                return 0;
                        }
                }
        }else if(!(strcmp(line->commands[0].argv[0], "jobs"))){
                for(int i = 0; i < num_proceso ; i++){
                        simb = " ";
                        if(i == num_proceso - 2){
                                simb = "-";
                        }else if(i == num_proceso - 1){
                                simb = "+";
                        }
                        printf("[%d]%s Running                %s\n", procesos[i].pid, simb,
procesos[i].comando);
                }
        }else if(!(strcmp(line->commands[0].argv[0], "fg"))){
                bool contiene;
                int pos;
                pid_t pidAux;
                if(line->commands[0].argc < 2){
                        waitpid(procesos[num_proceso - 1].pid, &status, 0);
                        procesos = (struct proceso *) realloc(procesos, num_proceso *
sizeof(struct proceso));
```

```
                        num_proceso--;
                    }else{
                        contiene = false;
                        pidAux = atoi(line->commands[0].argv[1]);

                        for(int i = 0; i < num_proceso; i++){
                            if(procesos[i].pid == pidAux){
                                contiene = true;
                                pos = i;
                            }
                        }
                        if(contiene){
                            waitpid(pidAux, &status, 0);
                            for(int i = pos; i < num_proceso - 1; i++){
                                procesos[i] = procesos[i + 1];
                            }
                            procesos = (struct proceso *) realloc(procesos,
num_proceso * sizeof(struct proceso));
                            num_proceso--;
                        }else{
                            printf("Ninguno de los procesos ejecutados en segundo
plano contiene el pid introducido.");
                        }
                    }
                }
```

```
            return num_proceso;
        }
```

Esta es la función UnicoMandatro, la cual se encarga de manejar el caso en el que el usuario introduzca por consola uno de los 3 mandatos especiales que se piden implementar, los cuales son “cd”, “jobs” y “fg”.

Lo primero es comprobar la línea que se ha escrito y se corresponde con alguno de los mandatos utilizando la función strcmp(). Para el caso de cd lo que se hará es, si no se recibe un directorio por defecto se cambiará al directorio $HOME. En caso contrario se comprobará si el directorio es válido y en ese caso se cambiará a dicho directorio con el uso de la función chdir().

Para el caso en el que la línea recibida sea “jobs” se imprimirá por pantalla un array de procesos previamente creado.

En caso de que se reciba por pantalla el comando “fg”, se recorre el mismo array que se recorre en jobs, en el cual estarán todos los procesos que se hayan añadido a background y se eliminarán.

```
void anadirProceso(pid_t pid, char * buf, int num_proceso){


        procesos = (struct proceso *) realloc(procesos, (num_proceso + 1) * sizeof(struct
proceso));
        procesos[num_proceso].pid = pid;
        strncpy(procesos[num_proceso].comando, buf, 1024);
}
```

Esta es la función AñadirProceso, el cual añadirá los procesos pasados a background a un array de procesos, el cual se utilizará para la función anteriormente explicada.

```
bool comandoPATH(char * comando){
        char *path_env = getenv("PATH");
   char *dir = strtok(path_env, ":");
        bool contiene = false;
   char path[256];

   while (dir != NULL) {

      snprintf(path, sizeof(path), "%s/%s", dir, comando);
      if (access(path, X_OK) == 0) {
         contiene = true;
      }

      dir = strtok(NULL, ":");
   }

        return contiene;
}
```

La función comandoPATH comprobará si la dirección que se le pasa es correcta para crear un fichero o leer de la dirección, para ello se utiliza access(path, X_OK) que verifica si el archivo existe y se puede acceder a él.

Manejador

```
void manejador(int sig){
        int status;
        int pid;
        int pos;
        bool contiene = false;

        if(sig == SIGCHLD){

                while ((pid = waitpid(-1, &status, WNOHANG)) > 0)
                {
                        if (WIFEXITED(status)){
                                contiene = false;

                                for(int i = 0; i < num_proceso; i++){
                                        if(procesos[i].pid == pid){
```

```
                                        contiene = true;
                                        pos = i;
                                }
                        }
                        if(contiene){
                                waitpid(pid, &status, 0);
                                for(int i = pos; i < num_proceso - 1; i++){
                                        procesos[i] = procesos[i + 1];
                                }
                                procesos = (struct proceso *) realloc(procesos,
num_proceso * sizeof(struct proceso));
                                num_proceso--;
                        }
                }
        }
}
```

El manejador permitirá controlar el funcionamiento de los hijos mediante señales, desde el cual se controla el estado que tendrán a lo largo de la ejecución.

Procesos padre- hijo.

La mejor forma de apreciar todo el funcionamiento y sintaxis utilizada será viendo la función main por completo.

```
int
main(void) {

        char buf[1024];
        tline * line;
        int i;
        pid_t pid;
        int fdinput, fdoutput;
        int status;
        procesos = (struct proceso *)malloc(num_proceso * sizeof(struct proceso));

        signal(SIGINT, SIG_IGN);
        signal(SIGQUIT, SIG_IGN);
        signal(SIGCHLD, manejador);
```

```
        printf("msh > ");
        while (fgets(buf, 1024, stdin)) {
                int only_spaces = 1;
                for (long unsigned int i = 0; i < strlen(buf); i++) {
                if (!isspace(buf[i])) {
                        only_spaces = 0;
                        break;
                    }
                }
                if ((strcmp(buf, "\n") != 0) && !only_spaces) {
                        line = tokenize(buf);

                        int fds[line->ncommands - 1][2];
                        if (!(strcmp(line->commands[0].argv[0], "exit"))){
                                break;
                        }
                        if (line==NULL) {
                                continue;
                        }

                        for (i=0; i<line->ncommands - 1; i++) {

                                pipe(fds[i]);

                        }

                        if (!(strcmp(line->commands[0].argv[0], "cd")) ||
!(strcmp(line->commands[0].argv[0], "jobs")) || (num_proceso > 0 &&
!(strcmp(line->commands[0].argv[0], "fg")))){

                                num_proceso = unicoMandato(line, num_proceso, status);

                        }else{

                                for (i=0; i<line->ncommands; i++) {

                                        pid = fork();
                                        if (pid < 0) { /* Error */
                                                fprintf(stderr, "Falló el fork()");
                                                exit(-1);
                                        }
                                        else if (pid == 0) { /* Proceso Hijo */
                                                if(!line->background){
                                                        signal(SIGINT, SIG_DFL);
                                                        signal(SIGQUIT, SIG_DFL);
                                                }
                                                if(line->ncommands > 1){
                                                        if(i == 0){

                                                                if (line->redirect_input != NULL) {
```

```
                                                    fdinput =
open(line->redirect_input, O_RDONLY);
                                                    dup2(fdinput,
STDIN_FILENO);
                                                    close(fdinput);
                                                }

                                                close(fds[i][READ_END]);
                                                dup2(fds[i][WRITE_END],
STDOUT_FILENO);
                                                close(fds[i][WRITE_END]);

execvp(line->commands[i].filename, line->commands[i].argv);
                                            }else if(i > 0 && i < (line->ncommands -
1)){

                                                close(fds[i][READ_END]);
                                                //Tiene que cerrar la escritura del
pipe anterior para asi leer de el
                                                dup2(fds[i-1][READ_END],
STDIN_FILENO);
                                                close(fds[i-1][READ_END]);

                                                //Tiene que cerrar la lectura del
pipe actual para asi escribir en el
                                                dup2(fds[i][WRITE_END],
STDOUT_FILENO);
                                                close(fds[i][WRITE_END]);


execvp(line->commands[i].filename, line->commands[i].argv);
                                            }else{

                                                dup2(fds[i-1][READ_END],
STDIN_FILENO);
                                                close(fds[i-1][READ_END]);

                                                if (line->redirect_output != NULL) {
                                                    //printf("redirección de
salida: %s\n", line->redirect_output);
                                                    fdoutput =
open(line->redirect_output, O_CREAT | O_WRONLY, 0777);

                                                    dup2(fdoutput,
STDOUT_FILENO);
                                                    //close(fdoutput);
                                                }

execvp(line->commands[i].filename, line->commands[i].argv);
                                            }
```

```
                                        }else {
                                                if (line->redirect_output != NULL) {
                                                        fdoutput =
open(line->redirect_output, O_CREAT | O_WRONLY, 0777);

                                                        dup2(fdoutput,
STDOUT_FILENO);
                                                        //close(fdoutput);
                                                }

execvp(line->commands[i].filename, line->commands[i].argv);
                                        }
                                        fprintf(stderr,"Se ha producido un error.\n");
                                        exit(1);
                                }
                                if(line->ncommands > 1){
                                        if (i == 0){
                                                close(fds[i][WRITE_END]);
                                        } else if(i > 0 && i < (line->ncommands - 1)){
                                                close(fds[i-1][READ_END]);
                                                close(fds[i][WRITE_END]);
                                        }
                                }
                        }

                        if (!line->background) {
                                for (i=0; i<line->ncommands - 1; i++) {
                                        close(fds[i][WRITE_END]);
                                        close(fds[i][READ_END]);
                                        wait (&status);
                                }
                                wait (&status);
                        }else{
                                anadirProceso(pid, buf, num_proceso);
                                num_proceso++;
                        }
                }
        }
        printf("msh > ");
    }
    return 0;
}
```

La función Main es la mas importante ya que controlará el funcionamiento de la mini-shell.

Para empezar, tendrá un bucle mientras que se encargará, mediante el uso de fgets(), de leer la línea actual de la consola y lo que se vaya escribiendo en ella.

Para controlar el comando y argumento que se está leyendo se utiliza el apartado de line->commands que forma parte de la librería parser.

Según el número de argumentos que recoja en cada línea se crearán un número específico de hijos mediante el uso de fork(). Es decir(for (i=0; i<line->ncommands; i++) { pid = fork();) tendrá en cuenta el número de comandos y realizará ese fork() commands veces, de esta forma cada proceso ejecuta un comando específico y se comunicaran mediante el uso de pipes() y sus "fd" correspondientes. Se deberá tener en cuenta cuando la pipes es de escritura o de lectura, cerrando para cada caso el lado de la pipe que no se vaya a usar utilizando close().

También se hace uso de "fdoutput = open(line->redirect_output, O_CREAT | O_WRONLY, 0777);" que se encargará de abrir un fichero para más adelante redireccionar la salida de la entrada estándar al fichero o viceversa.

Otras funciones a tener en cuenta son, por ejemplo, el uso de SIG_IGN y SIG_DFL, los cuales permiten ignorar o no, según lo especifiquemos, el uso de comandos como pueden ser CNTRL+C.

Una parte importante del main es la comprobación de los comandos que el usuario quiera pasar a background, llamando en el caso de que así sea a la función AñadirProceso, para poder guardar esos procesos y poder mostrarlos así en el jobs. Para ellos se utiliza la estructura condicional " *if (!line->background) ".*

Discusión y conclusiones

Como se mencionó en el resumen, en esta práctica hemos podido estudiar en profundidad el uso y creación de diferentes procesos mediante el uso de fork y cómo funciona la comunicación entre ellos.

En comparación a la primera práctica, esta ha tenido un nivel de dificultad mucho más elevado, además de llevarnos más horas de trabajo e investigación para entender bien el uso de cada apartado, llegando incluso al doble de tiempo empleado para la primera.

Han existido varias complicaciones también a la hora de comunicar bien todos los procesos y usar bien el manejador. Pero nos ha parecido un reto interesante que nos ha ayudado sobre todo a estudiar de cara al examen.