CodigoPractica2.pdf

sirenejota

Sistemas Operativos

2º Grado en Ingeniería de Computadores

Escuela Técnica Superior de Ingeniería Informática. Campus de Móstoles
Universidad Rey Juan Carlos


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "parser.h"
#include <sys/wait.h>
#include <fcntl.h>
#include <stdbool.h>
#include <string.h>
#include <signal.h>


#define READ_END 0
#define WRITE_END 1

// Estructura de un trabajo //

struct job{
    pid_t pid;
    char buf[1024];
};

// Puntero al PID en foreground (sin reserva de memoria aún) //

pid_t *foreground_pid;

// Manejadores de las señales SIGINT y SIGQUIT //

void sigint_handler(int signal){
int i = 0;
while (foreground_pid[i] > 0){
    kill (foreground_pid[i], signal);
    i++;
}
}

void sigquit_handler(int signal){
int i = 0;
while (foreground_pid[i]>0){
    kill(foreground_pid[i], signal);
    i++;
}

}

// Funciones para cd, jobs y fg //

int doCd (int argc, char *argv[]){

    char *path;
    char buffer[1024];

    /// ELECCIÓN DEL DIRECTORIO ///

```

```
54      if (argc == 1){ //Si el usuario solo ha escrito "cd"
55
56          path = getenv("HOME");
57
58      } else if (argc == 2){ //Si el usuario ha escrito un directorio
59
60          path = argv[1];
61
62      } else { //Si el usuario ha escrito argumentos de más
63
64          perror("Error.Se han introducido demasiados argumentos\nUso: cd
               o cd DIRECTORIO\n");
65          return EXIT_FAILURE;
66      }
67
68      /// CAMBIO DE DIRECTORIO ///
69
70      if(chdir(path) < 0){
71
72          perror("Error. Fallo al cambiar de directorio");
73          return EXIT_FAILURE;
74      }
75
76      /// IMPRESIÓN DEL DIRECTORIO PARA EL CONTROL ///
77
78      printf("Se ha establecido el directorio actual en: %s\n", getcwd
          (buffer,1024));
79      return EXIT_SUCCESS;
80  }
81
82  void showJobs (int *numjobs, struct job jobs[]){
83
84      int i,j;
85      pid_t result;
86      int status;
87
88      /// Muestra de trabajos ///
89
90      for (i = 0; i<*numjobs; i++){
91
92          result = waitpid(jobs[i].pid, &status, WNOHANG);
93
94          if(result == 0){ // El proceso se está ejecutando
95
96              if((i+1) == *numjobs){ //Si el trabajo actual es el último
                  del array
97
98                  printf("[%d]+\tEjecutando\t%s\n", i, jobs[i].buf); //
                      "+" indica que es el último trabajo.
99
100             } else if ((i+2) == *numjobs) { //Si el trabajo actual es
                  el penúltimo del array
101
```

```
102                 printf("[%d]-\tEjecutando\t%s\n", i, jobs[i].buf); //
                      "-" indica que es el penúltimo trabajo.
103
104               }else { //Si el trabajo actual no es ni el último ni el
                    penúltimo.
105
106                 printf("[%d]\tEjecutando\t%s\n", i, jobs[i].buf);
107
108             }
109
110         } else if(result > 0 ){ // El proceso ha terminado
111
112             if((i+1) == *numjobs){ //Si el trabajo actual es el último
                    del array
113
114                 printf("[%d]+\tTerminado\t%s\n", i, jobs[i].buf);
115
116             } else if ((i+2) == *numjobs) { //Si el trabajo actual es
                    el penúltimo del array
117
118                 printf("[%d]-\tTerminado\t%s\n", i, jobs[i].buf);
119
120             }else { //Si el trabajo actual no es ni el último ni el
                    penúltimo.
121
122                 printf("[%d]\tTerminado\t%s\n", i, jobs[i].buf);
123
124             }
125
126         } else { //El proceso ha ocasionado un error
127
128              if((i+1) == *numjobs){ //Si el trabajo actual es el último
                    del array
129
130                 printf("[%d]+\tError en waitpid\t%s\n", i, jobs
                      [i].buf);
131
132             } else if ((i+2) == *numjobs) { //Si el trabajo actual es
                    el penúltimo del array
133
134                 printf("[%d]-\tError en waitpid\t%s\n", i, jobs
                      [i].buf);
135
136             }else { //Si el trabajo actual no es ni el último ni el
                    penúltimo.
137
138                 printf("[%d]\tError en waitpid\t%s\n", i, jobs
                      [i].buf);
139
140             }
141         }
142
143 }
```

```
144
145   /*La terminal, cuando se ejecuta el comando jobs, muestra los
        terminados y, después, los elimina*/
146
147         // Eliminación de los procesos terminados y erróneos//
148
149         for(i = 0; i < *numjobs; i++){
150         result = waitpid(jobs[i].pid, &status, WNOHANG);
151         if(result!=0){
152             for(j=i; j < *numjobs-1; j++){
153             jobs[i] = jobs [i+1];
154             }
155         (*numjobs)--;
156         i--;
157
158         }
159     }
160 }
161
162 void executeForeground(struct job jobs[], int* numJobs, char *argv[]){
163     int jobIndex, i;
164
165     if (argv[1] == NULL){ // Si solo se ha introducido "fg", sin
        argumentos
166
167         waitpid(jobs[(*numJobs)-1].pid, NULL, WUNTRACED); // Se espera
          al último trabajo de la lista
168
169         (*numJobs)--; // Se reduce el número de trabajos de la lista.
170
171     } else { // Si se ha introducido un argumento después de "fg"
172
173         jobIndex = atoi(argv[1]);
174         waitpid(jobs[jobIndex].pid, NULL, WUNTRACED); // Se espera al
          trabajo especificado.
175
176         for(i = jobIndex; i < (*numJobs) -1; i++){ // Se desplazan los
          trabajos restantes
177             jobs[i] = jobs [i+1];
178         }
179         (*numJobs)--; // Se reduce el número de trabajos de la lista.
180  }
181 }
182
183
184 int main (void){
185
186     // Configuración de los manejadores de señales:
187
188     signal (SIGINT, sigint_handler);
189     signal (SIGQUIT, sigquit_handler);
190
191     // Creación de variables varias:
```

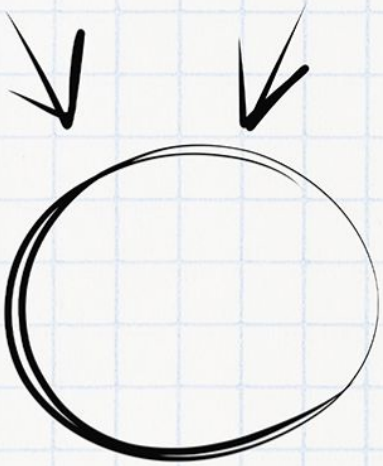
```
192
193     char buf[1024];
194     int numjobs = 0;
195     tline *line;
196
197     // Creación de los file descriptor:
198
199     int stdout_fd = dup(STDOUT_FILENO);
200     int stdin_fd = dup (STDIN_FILENO);
201     int stderr_fd = dup (STDERR_FILENO);
202
203     int i, j;
204     int fd_in, fd_out, fd_err;
205
206     // Background:
207
208     int background = 0;
209     struct job jobs[20]; //Array de trabajos en background
210
211     // Pipes:
212
213     int **pipes;
214
215 /////////// Inicio del programa en sí ///////////
216
217 printf("\n>> ");
218
219 while (fgets(buf,1024,stdin)!= NULL){ // Mientras se reciban comandos
      de entrada
220
221     pid_t pid;
222
223     line = tokenize(buf); // Dividimos el comando en partes (tokens)
224
225     pipes = malloc(sizeof(int *) * (line->ncommands - 1)); // Asignamos
          memoria dinámica a in array de pipes (1 por comando)
226
227     if (pipes == NULL){
228         perror("Fichero: Error. Fallo en la asignación de memoria de
              los pipes.\n");
229         exit(EXIT_FAILURE);
230     }
231
232     if(line == NULL){ // Si line es nulo significa que no ha podido
          analizarse la línea de comandos
233         continue;
234     }
235
236     ///// REDIRECCIONES /////
237
238     if (line->redirect_input != NULL){ //Si al cambiar la entrada
          estándar, line no es nulo, continuamos.
239
```

Imagínate aprobando el examen

Necesitas tiempo y concentración

Planes	PLAN TURBO	PLAN PRO	PLAN PRO+
Descargas sin publi al mes	10	40	80
Elimina el video entre descargas	✓	✓	✓
Descarga carpetas	✗	✓	✓
Descarga archivos grandes	✗	✓	✓
Visualiza apuntes online sin publi	✗	✓	✓
Elimina toda la publi web	✗	✗	✓
Precios Anual	0,99 € / mes	3,99 € / mes	7,99 € / mes

Ahora que puedes conseguirlo, ¿Qué nota vas a sacar?



WUOLAH

```
240         printf("Redirigiendo la entrada: %s\n", line->redirect_input);
241         fd_in = open(line->redirect_input, O_RDONLY); // Se abre el archivo redirigido en modo lectura.
242
243         if (fd_in < 0){
244
245             perror("Fichero: Error. Fallo al abrir el archivo de entrada.\n");
246             exit(EXIT_FAILURE);
247         }
248
249         // Solo se llegará a esta parte del código si no ha fallado, por lo que podemos terminar de redirigir la entrada.
250
251         dup2(fd_in, STDIN_FILENO);
252         close(fd_in);
253     }
254
255     if (line->redirect_output != NULL) //Si al cambiar la salida estándar, line no es nulo, continuamos.
256     {
257         printf("Redirigiendo la salida: %s\n", line->redirect_output);
258         fd_out = open(line->redirect_output, O_WRONLY | O_CREAT | O_TRUNC, 0600); // Lo abrimos en modo escritura, si no existe lo creamos y, si existe, lo truncamos.
259
260         if (fd_out < 0){
261
262             perror("Fichero: Error. Fallo al abrir el archivo de salida.\n%s\n");
263             exit(EXIT_FAILURE);
264         }
265
266         // Solo se llegará a esta parte del código si no ha fallado, por lo que podemos terminar de redirigir la salida.
267
268         dup2(fd_out, STDOUT_FILENO);
269         close(fd_out);
270     }
271
272     if(line->redirect_error != NULL){ //Si al cambiar la salida de errores, line no es nulo, continuamos.
273
274         printf("Redirigiendo la salida de errores: %s\n", line->redirect_error);
275         fd_err = open(line->redirect_error, O_WRONLY | O_CREAT | O_TRUNC, 0600); // Lo abrimos en modo escritura, si no existe lo creamos y, si existe, lo truncamos.
276
277         if (fd_err < 0){
278
279             perror("Error. Fallo al abrir el archivo de salida de errores.\n%s\n");
```

```
280             exit(EXIT_FAILURE);
281         }
282
283         // Solo se llegará a esta parte del código si no ha fallado,
              por lo que podemos terminar de redirigir la salida de
              errores.
284
285         dup2(fd_err, STDERR_FILENO);
286         close(fd_err);
287
288     }
289
290     ///// DECISIÓN DE BACKGROUND /////
291
292     if (line->background){ // Si background es verdadero, el comando se
          ejecutará como tal.
293
294     printf("El comando introducido se ejecutará en background\n");
295
296     background = 1;
297     strcpy(jobs[numjobs].buf,buf); //Copiamos el comando en el campo
          buf del job correspondiente en el array jobs.
298     numjobs++;
299     }else{ // Cuando background es falso
300         foreground_pid = malloc(sizeof(pid_t*)*line->ncommands); //Se
              asigna memoria dinámica a foreground_pid, de manera que
              podemos leer ncommands pids.
301     }
302
303
304     ///// ACTIVACIÓN DE PIPES /////
305
306     for (i = 0; i<line->ncommands -1; i++){
307
308         pipes[i] = malloc(sizeof(int)*2); //Reserva de memoria para dos
              enteros.
309         if(pipe(pipes[i]) < 0){
310             perror("Error. Fallo al crear una tubería.\n%s\n");
311             exit(EXIT_FAILURE);
312         }
313     }
314
315     ///// RECORRIDO DE LOS COMANDOS ALMACENADOS EN LINE /////
316
317     for (i = 0; i < line->ncommands; i++){
318
319         printf("Comando número %d: %s\n", i, line->commands
              [i].filename);
320
321         for(j = 0; j < line->commands[i].argc; j++){
322             printf("Argumento número %d: %s\n ", j, line->commands
                  [i].argv[j]);
323         }
```

```
324
325         if(!strcmp(line->commands[i].argv[0], "jobs")){ // Si el comando es jobs
326
327             showJobs(&numjobs,jobs);
328
329         }else if (!strcmp(line->commands[i].argv[0], "cd")){ // Si el comando es cd
330
331             doCd(line->commands[i].argc, line->commands[i].argv);
332
333         }else if(!strcmp(line->commands[i].argv[0], "fg")){ // Si el comando es fg
334
335             executeForeground(jobs, &numjobs, line->commands[i].argv);
336
337         } else { // No es ni jobs, ni cd, ni fg.
338
339         // Creación del proceso hijo //
340
341             pid = fork();
342
343             if (pid > 0){ /*Proceso padre*/
344
345             //Gestión de pipes
346
347                 if (i != 0){ /*Si el comando no es el primero de la lista de comandos*/
348                     close(pipes[i-1][WRITE_END]);
349                     close(pipes[i-1][READ_END]);
350                 }
351
352                 if ((i+1) != line->ncommands) { /*Si el comando no es el último de la lista*/
353                     close(pipes[i][WRITE_END]);
354                 }
355
356                 if (background) /*Si está en background se realiza directamente*/
357                 {
358                     jobs[numjobs-1].pid = pid;
359                     waitpid(pid, NULL, WNOHANG);
360                     printf("[%d] %d", numjobs, pid);
361
362                 } else { /*Si está en foreground espera a que el hijo acabe*/
363
364                     foreground_pid[i] = pid;
365                     waitpid(pid, NULL, 0);
366                 }
367
368             } else if (pid == 0){ /*Proceso hijo*/
369
```

```
370                 if(i!=0){
371
372                     close(pipes[i-1][WRITE_END]);
373                     dup2(pipes[i-1][READ_END], STDIN_FILENO);
374                     close(pipes[i-1][READ_END]);
375                 }
376
377                 if((i+1) != line->ncommands){
378
379                     dup2(pipes[i][WRITE_END], STDOUT_FILENO);
380                     close(pipes[i][WRITE_END]);
381                     close(pipes[i][READ_END]);
382                 }
383
384                 execvp(line->commands[i].argv[0], line->commands
                      [i].argv);
385                 fprintf(stderr, "Mandato: Error. No se encuentra el
                      mandato: %s\n", line->commands[i].argv[0]);
386                 exit(EXIT_FAILURE);
387
388             } else { /*Error*/
389                 perror("Error. Fallo en el fork()");
390                 exit(EXIT_FAILURE);
391             }
392         }
393     }
394
395     if (!background){
396
397         free(foreground_pid); //Se libera
398     }
399
400     // REINICIO DE VARIABLES Y ENTRADAS //
401
402     background = 0;
403     dup2(stdin_fd, STDIN_FILENO);
404     dup2(stdout_fd, STDOUT_FILENO);
405     dup2(stderr_fd, STDERR_FILENO);
406
407     // LIBERACIÓN DE MEMORIA //
408
409     for(i = 0; i < line-> ncommands -1; i++){
410         free (pipes[i]);
411     }
412     printf("\n>> ");
413     }
414     return 0;
415 }
```