~\Desktop\ING DEL SOFTWARE\SOFTWARE 2_2\SISTEMAS OPERATIVOS\Prácticas\Práctica 4\prShellAmpliaciones\Shell_project.c

```
/**
 * Decena Giménez, Macorís
 * Software A
 *
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)

**/

#include <string.h>
#include <pthread.h>
#include "job_control.h"   // remember to compile with module job_control.c

#define MAX_LINE 256 /* 256 chars per line, per command, should be enough. */
job * listaTareas;

// Parse redirections operators '<' '>' once args structure has been built
// Call immediately after get_commad()
//    get_command(..);
//    char *file_in, *file_out;
//    parse_redirections(args, &file_in, &file_out);
//
// For a valid redirection, a blank space is required before and after
// redirection operators '<' or '>'
void parse_redirections(char **args, char **file_in, char **file_out)
{
    *file_in = NULL;
    *file_out = NULL;
    char **args_start = args;
    while (*args)
    {
        int is_in = !strcmp(*args, "<");
        int is_out = !strcmp(*args, ">");
        if (is_in || is_out)
        {
            args++;
            if (*args)
            {
                if (is_in)
                    *file_in = *args;
                if (is_out)
                    *file_out = *args;
                char **aux = args + 1;
                while (*aux)
                {
```

```
56                      *(aux - 2) = *aux;
57                      aux++;
58                  }
59                  *(aux - 2) = NULL;
60                  args--;
61              }
62              else
63              {
64                  /* Syntax error */
65                  fprintf(stderr, "syntax error in redirection\n");
66                  args_start[0] = NULL; // Do nothing
67              }
68          }
69          else
70          {
71              args++;
72          }
73      }
74  }
75
76  void manejador(int signal) {
77      block_SIGCHLD();
78      job * tarea;
79      int status;
80      int info;
81      int pid_wait = 0;
82      int pid_respawnable = 0;
83      enum status status_res;
84
85      for (int i = list_size(listaTareas); i >= 1; i--) {
86          tarea = get_item_bypos(listaTareas, i);
87          pid_wait = waitpid(tarea->pgid, &status, WUNTRACED | WNOHANG | WCONTINUED);
88
89          if (pid_wait == tarea->pgid) {
90              status_res = analyze_status(status, &info);
91
92              if (status_res == SUSPENDED) {
93                  if (tarea->state == RESPAWNABLE)
94                      printf("\nRespawnable pid: %d, command: %s, %s, info: %d\n\n", tarea-
    >pgid, tarea->command, status_strings[status_res], info);
95
96                  else
97                      printf("\nBackground pid: %d, command: %s, %s, info: %d\n\n", tarea-
    >pgid, tarea->command, status_strings[status_res], info);
98
99                  tarea->state = STOPPED;
100             }
101
102             else if (status_res == EXITED || status_res == SIGNALED) {
103                 if (info != 255) {
104                     if (tarea->state == RESPAWNABLE) {  // Amp 1
105                         printf("\nRespawnable pid: %d, command: %s, %s, info: %d\n",
    tarea->pgid, tarea->command, status_strings[status_res], info);
106
107
108                         pid_respawnable = fork();
109
110                         if (pid_respawnable < 0) {
111                             printf("Could not create respawnable process.");
112                         }
113                         else if (pid_respawnable == 0) {    // Hijo
```

```
114                             new_process_group(getpid());
115                             restore_terminal_signals();
116                             execvp(tarea->command, tarea->args);
117 
118                             printf("\nError, command not found\n\n");
119 
120                             exit(-1);   // Si execvp no se ejecuta, es porque el comando es erroneo.
121                         }
122                         else {      // Padre
123                             printf("\nRespawnable job running again... pid: %d, command: %s\n", pid_respawnable, tarea->command);
124                             new_process_group(pid_respawnable);
125                             tarea->pgid = pid_respawnable;
126                         }
127                     }
128                     else {
129                         printf("\nBackground pid: %d, command: %s, %s, info: %d\n\n", tarea->pgid, tarea->command, status_strings[status_res], info);
130                         delete_job(listaTareas, tarea);
131                     }
132                 }
133                 else {
134                     delete_job(listaTareas, tarea);
135                 }
136             }
137 
138             else if (status_res == CONTINUED)
139             {
140                 printf("\nBackground pid: %d, command: %s, %s, info: %d\n\n", tarea->pgid, tarea->command, status_strings[status_res], info);
141 
142                 if (tarea->state != RESPAWNABLE)
143                     tarea->state = BACKGROUND;
144             }
145         }
146     }
147     unblock_SIGCHLD();
148 }
149 
150 void* timeoutFunc(void* arg) {      // Amp 2
151     int segundos = *((int *) arg);
152     pid_t pid = *(pid_t *)((int*)arg + 1);
153 
154     sleep(segundos);
155     kill(pid, SIGKILL);
156 
157     return NULL;
158 }
159 
160 // -----------------------------------------------------------------------
161 //                            MAIN
162 // -----------------------------------------------------------------------
163 
164 int main(void)
165 {
166     char inputBuffer[MAX_LINE]; /* buffer to hold the command entered */
167     int background;             /* equals 1 if a command is followed by '&' */
168     int respawnable;            /* equals 1 if a command is followed by '+' */  // Amp 1
169     char *args[MAX_LINE/2];     /* command line (of 256) has max of 128 arguments */
170     // probably useful variables:
```

```c
    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() */

    job * tarea;
    int primerPlano = 0;

    ignore_terminal_signals();
    signal(SIGCHLD, manejador);
    listaTareas = new_list("Tareas");

    char *file_in = NULL;
    char *file_out = NULL;
    FILE *fin = stdin;
    FILE *fout = stdout;

    // Amp 2
    int time;
    int timeout;
    pthread_t timeout_thread;
    int error_thread;

    // Amp 3
    int mask;

    while (1)   /* Program terminates normally inside get_command() after ^D is typed*/
    {
        timeout = 0;
        mask = 0;
        printf("COMMAND->");
        fflush(stdout);
        get_command(inputBuffer, MAX_LINE, args, &background, &respawnable);  /* get next command */
        parse_redirections(args, &file_in, &file_out);

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

        /* the steps are:
             (1) fork a child process using fork()
             (2) the child process will invoke execvp()
             (3) if background == 0, the parent will wait, otherwise continue
             (4) Shell shows a status message for processed command
             (5) loop returns to get_commnad() function
        */

        if (strcmp(args[0], "cd") == 0) {
            int dirValido = chdir(args[1]);
            if (dirValido == -1)
                printf("\nError, directory not found\n\n");
            continue;
        }

        if (strcmp(args[0], "jobs") == 0) {
            block_SIGCHLD();
            print_job_list(listaTareas);
            unblock_SIGCHLD();
            continue;
        }

```

```
230         if (strcmp(args[0], "bg-jobs") == 0) {  // Amp extra
231             block_SIGCHLD();
232             print_bg_job_list(listaTareas);
233             unblock_SIGCHLD();
234             continue;
235         }
236 
237         if (strcmp(args[0], "stp-jobs") == 0) { // Amp extra
238             block_SIGCHLD();
239             print_stp_job_list(listaTareas);
240             unblock_SIGCHLD();
241             continue;
242         }
243 
244         if (strcmp(args[0], "rsp-jobs") == 0) { // Amp extra
245             block_SIGCHLD();
246             print_rsp_job_list(listaTareas);
247             unblock_SIGCHLD();
248             continue;
249         }
250 
251         if (strcmp(args[0], "fg") == 0) {
252             block_SIGCHLD();
253 
254             int pos = 1;
255             if (args[1] != NULL) {
256                 pos = atoi(args[1]);
257             }
258 
259             tarea = get_item_bypos(listaTareas, pos);
260 
261             if (tarea != NULL) {
262                 primerPlano = 1;
263                 set_terminal(tarea->pgid);
264                 if (tarea->state == STOPPED) {
265                     killpg(tarea->pgid, SIGCONT);
266                 }
267                 pid_fork = tarea->pgid;
268                 strcpy(args[0], tarea->command);
269                 delete_job(listaTareas, tarea);
270             }
271 
272             unblock_SIGCHLD();
273         }
274 
275         if (strcmp(args[0], "bg") == 0) {
276             block_SIGCHLD();
277             int pos = 1;
278             if (args[1] != NULL) {
279                 pos = atoi(args[1]);
280             }
281 
282             tarea = get_item_bypos(listaTareas, pos);
283 
284             if (tarea != NULL && (tarea->state == STOPPED || tarea->state == RESPAWNABLE))
{   // Amp 1
285                 tarea->state = BACKGROUND;
286                 killpg(tarea->pgid, SIGCONT);
287             }
288 
```

```
289             unblock_SIGCHLD();
290             continue;
291         }
292 
293         if (strcmp(args[0], "rsp") == 0) {  // Amp extra
294             block_SIGCHLD();
295             int pos = 1;
296             if (args[1] != NULL) {
297                 pos = atoi(args[1]);
298             }
299 
300             tarea = get_item_bypos(listaTareas, pos);
301 
302             if (tarea != NULL && (tarea->state == STOPPED || tarea->state == BACKGROUND))
    {
303                 tarea->state = RESPAWNABLE;
304                 killpg(tarea->pgid, SIGCONT);
305             }
306 
307             unblock_SIGCHLD();
308             continue;
309         }
310 
311         if (strcmp(args[0], "alarm-thread") == 0) {     // Amp 2
312             if (args[1] != NULL && args[2] != NULL) {
313                 timeout = 1;
314 
315                 time = atoi(args[1]);
316 
317                 if (time == 0) {
318                     printf("Error, illegal argument (not a number): %s\n\n", args[1]);
319                     continue;
320                 }
321 
322                 int i=0;
323 
324                 while (args[i+2] != NULL) {
325                     args[i] = strdup(args[i+2]);
326                     args[i+2] = NULL;
327                     i++;
328                 }
329                 args[i] = NULL;
330             }
331             else {
332                 printf("Error, not enough arguments for alarm-thread command\n\n");
333                 continue;
334             }
335         }
336 
337         if (strcmp(args[0], "mask") == 0) {     // Amp 3
338             if (args[1] != NULL && args[2] != NULL && args[3] != NULL) {
339                 mask = atoi(args[1]);
340 
341                 if (mask == 0) {
342                     printf("Error, illegal argument (not a number): %s\n\n", args[1]);
343                     continue;
344                 }
345 
346                 if (strcmp(args[2], "-c") != 0) {
```

```
347                     printf("Error, illegal argument (does not equal -c): %s\n\n",
    args[2]);
348                     continue;
349                 }
350 
351                 int i=0;
352 
353                 while (args[i+3] != NULL) {
354                     args[i] = strdup(args[i+3]);
355                     args[i+3] = NULL;
356                     i++;
357                 }
358                 args[i] = NULL;
359                 args[i+1] = NULL;
360 
361                 // Las 2 siguientes lineas las he incluido porque en el enunciado se dice
362                 // que el comando ejecutado con mask será en foreground.
363                 // Al no especificar si debe ignorar los posibles +/& añadidos al final,
364                 // yo he interpretado que SÍ debe ignorarlos.
365                 // Para no ignorarlos, solo habría que borrar estas 2 líneas.
366                 background = 0;
367                 respawnable = 0;
368             }
369             else {
370                 printf("Error, not enough arguments for mask command\n\n");
371                 continue;
372             }
373         }
374 
375         if (primerPlano == 0)
376             pid_fork = fork();
377 
378         if (pid_fork != 0) {    // Proceso padre
379 
380             new_process_group(pid_fork);
381 
382             if (timeout) {      // Amp 2
383                 int thread_args[2];
384                 thread_args[0] = time;
385                 thread_args[1] = pid_fork;
386 
387                 error_thread = pthread_create(&timeout_thread, NULL, timeoutFunc,
    (void*)thread_args);
388                 if (error_thread != 0) {
389                     printf("Error, thread could not be created");
390                     exit(-1);
391                 }
392 
393                 error_thread = pthread_detach(timeout_thread);
394                 if (error_thread != 0) {
395                     printf("Error, thread could not be detached");
396                     exit(-1);
397                 }
398 
399                 timeout = 0;
400             }
401 
402             if (background == 0 && respawnable == 0) {  // Primer plano
403                 set_terminal(pid_fork);
404 
```

```c
                pid_wait = waitpid(pid_fork, &status, WUNTRACED);

                set_terminal(getpid());

                if (pid_wait == -1) {
                    printf("\n\nError en waitpid");
                    exit(-1);
                }

                else if (pid_fork == pid_wait) {
                    status_res = analyze_status(status, &info);

                    if (status_res == SUSPENDED) {
                        block_SIGCHLD();
                        tarea = new_job(pid_fork, args[0], STOPPED);
                        add_job(listaTareas, tarea);
                        printf("\nForeground pid: %d, command: %s, %s, info: %d\n\n", pid_fork, args[0], status_strings[status_res], info);
                        unblock_SIGCHLD();
                    }

                    else if (status_res == SIGNALED) {
                        printf("\nForeground pid: %d, command: %s, %s, info: %d\n\n", pid_fork, args[0], status_strings[status_res], info);
                    }

                    else if (status_res == EXITED) {
                        if (info != 255)
                            printf("\nForeground pid: %d, command: %s, %s, info: %d\n\n", pid_fork, args[0], status_strings[status_res], info);
                    }
                }

                primerPlano = 0;
            }

            else if (background == 1) { // Segundo plano
                block_SIGCHLD();
                tarea = new_job(pid_fork, args[0], BACKGROUND);
                add_job(listaTareas, tarea);
                printf("\nBackground job running... pid: %d, command: %s\n\n", pid_fork, args[0]);
                unblock_SIGCHLD();
            }

            else if ((respawnable == 1) && (!(strcmp(args[0], "sleep") == 0 && args[1] == NULL))) {   // Respawnable (Amp 1)
                block_SIGCHLD();
                tarea = new_job(pid_fork, args[0], RESPAWNABLE);

                add_respawnable_job(listaTareas, tarea, args);
                printf("\nRespawnable job running... pid: %d, command: %s\n\n", pid_fork, args[0]);
                unblock_SIGCHLD();
            }

        }

        else {  // Proceso hijo
            new_process_group(getpid());

```

```
460             if (mask > 0) {     // Amp 3
461                 block_signal(mask, 1);
462             }
463 
464             if (background == 0 && respawnable == 0) { // Primer plano
465                 set_terminal(getpid());
466             }
467 
468             restore_terminal_signals();
469 
470             // Abrir los ficheros y hacer el dup
471 
472             fin = stdin;
473             if (file_in)
474             {
475                 fin = fopen(file_in, "r");
476                 if (!fin)
477                 {
478                     fprintf(stderr, "Error opening file %s for reading\n", file_in);
479                     return 1;
480                 }
481             }
482 
483             fout = stdout;
484             if (file_out)
485             {
486                 fout = fopen(file_out, "w");
487                 if (!fout)
488                 {
489                     fprintf(stderr, "Error opening file %s for writing\n", file_out);
490                     return 1;
491                 }
492             }
493 
494             // Redirection
495             dup2(fileno(fin), fileno(stdin));
496             dup2(fileno(fout), fileno(stdout));
497 
498             execvp(args[0], args);
499 
500             // Restore standard input and output
501             //dup2(fileno(stdin), fileno(fin));
502             //dup2(fileno(stdout), fileno(fout));
503 
504             dup2(STDERR_FILENO, STDOUT_FILENO);
505 
506             // Close file pointers if they were opened
507             if (file_in)
508                 fclose(fin);
509             if (file_out) {
510                 fclose(fout);
511             }
512 
513             printf("\nError, command not found: %s\n\n", args[0]);
514 
515             exit(-1);   // Si execvp no se ejecuta, es porque el comando es erroneo.
516         }
517 
518     } // end while
519 }
```