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

```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 "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)
                {
                    *(aux - 2) = *aux;
```

```
                    aux++;
                }
                *(aux - 2) = NULL;
                args--;
            }
            else
            {
                /* Syntax error */
                fprintf(stderr, "syntax error in redirection\n");
                args_start[0] = NULL; // Do nothing
            }
        }
        else
        {
            args++;
        }
    }
}

void manejador(int signal) {
    block_SIGCHLD();
    job * tarea;
    int status;
    int info;
    int pid_wait = 0;
    enum status status_res;

    for (int i = list_size(listaTareas); i >= 1; i--) {
        tarea = get_item_bypos(listaTareas, i);
        pid_wait = waitpid(tarea->pgid, &status, WUNTRACED | WNOHANG | WCONTINUED);

        if (pid_wait == tarea->pgid) {
            status_res = analyze_status(status, &info);

            if (status_res == SUSPENDED) {
                printf("\nBackground pid: %d, command: %s, %s, info: %d\n\n", tarea->pgid,
tarea->command, status_strings[status_res], info);

                tarea->state = STOPPED;
            }

            else if (status_res == EXITED || status_res == SIGNALED) {
                if (info != 255)
                    printf("\nBackground pid: %d, command: %s, %s, info: %d\n\n", tarea-
>pgid, tarea->command, status_strings[status_res], info);

                delete_job(listaTareas, tarea);
            }

            else if (status_res == CONTINUED)
            {
                printf("\nBackground pid: %d, command: %s, %s, info: %d\n\n", tarea->pgid,
tarea->command, status_strings[status_res], info);
                tarea->state = BACKGROUND;
            }
        }
    }
    unblock_SIGCHLD();
}

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

```c
//                          MAIN
// -----------------------------------------------------------------------

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() */

    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;

    while (1)   /* Program terminates normally inside get_command() after ^D is typed*/
    {
        printf("COMMAND->");
        fflush(stdout);
        get_command(inputBuffer, MAX_LINE, args, &background);  /* 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;
        }

        if (strcmp(args[0], "fg") == 0) {
            block_SIGCHLD();

```

```
174             int pos = 1;
175             if (args[1] != NULL) {
176                 pos = atoi(args[1]);
177             }
178 
179             tarea = get_item_bypos(listaTareas, pos);
180 
181             if (tarea != NULL) {
182                 primerPlano = 1;
183                 set_terminal(tarea->pgid);
184                 if (tarea->state == STOPPED) {
185                     killpg(tarea->pgid, SIGCONT);
186                 }
187                 pid_fork = tarea->pgid;
188                 strcpy(args[0], tarea->command);
189                 delete_job(listaTareas, tarea);
190             }
191 
192             unblock_SIGCHLD();
193         }
194 
195         if (strcmp(args[0], "bg") == 0) {
196             block_SIGCHLD();
197             int pos = 1;
198             if (args[1] != NULL) {
199                 pos = atoi(args[1]);
200             }
201 
202             tarea = get_item_bypos(listaTareas, pos);
203 
204             if (tarea != NULL && tarea->state == STOPPED) {
205                 tarea->state = BACKGROUND;
206                 killpg(tarea->pgid, SIGCONT);
207             }
208 
209             unblock_SIGCHLD();
210             continue;
211         }
212 
213         if (primerPlano == 0)
214             pid_fork = fork();
215 
216         if (pid_fork != 0) {    // Proceso padre
217 
218             new_process_group(pid_fork);
219 
220             if (background != 1) {  // Primer plano
221                 set_terminal(pid_fork);
222 
223                 pid_wait = waitpid(pid_fork, &status, WUNTRACED);
224 
225                 set_terminal(getpid());
226 
227                 if (pid_wait == -1) {
228                     printf("\n\nError en waitpid");
229                     exit(-1);
230                 }
231 
232                 else if (pid_fork == pid_wait) {
233                     status_res = analyze_status(status, &info);
```

```
234
235                     if (status_res == SUSPENDED) {
236                         block_SIGCHLD();
237                         tarea = new_job(pid_fork, args[0], STOPPED);
238                         add_job(listaTareas, tarea);
239                         printf("\nForeground pid: %d, command: %s, %s, info: %d\n\n",
    pid_fork, args[0], status_strings[status_res], info);
240                         unblock_SIGCHLD();
241                     }
242
243                     else if (status_res == SIGNALED) {
244                         printf("\nForeground pid: %d, command: %s, %s, info: %d\n\n",
    pid_fork, args[0], status_strings[status_res], info);
245                     }
246
247                     else if (status_res == EXITED) {
248                         if (info != 255)
249                             printf("\nForeground pid: %d, command: %s, %s, info: %d\n\n",
    pid_fork, args[0], status_strings[status_res], info);
250                     }
251                 }
252
253                 primerPlano = 0;
254             }
255
256             else {  // Segundo plano
257                 block_SIGCHLD();
258                 tarea = new_job(pid_fork, args[0], BACKGROUND);
259                 add_job(listaTareas, tarea);
260                 printf("\nBackground job running... pid: %d, command: %s\n\n", pid_fork,
    args[0]);
261                 unblock_SIGCHLD();
262             }
263
264         }
265
266         else {  // Proceso hijo
267             new_process_group(getpid());
268
269             if (background != 1) { // Primer plano
270                 set_terminal(getpid());
271             }
272
273             restore_terminal_signals();
274
275             // Abrir los ficheros y hacer el dup
276
277             fin = stdin;
278             if (file_in)
279             {
280                 fin = fopen(file_in, "r");
281                 if (!fin)
282                 {
283                     fprintf(stderr, "Error opening file %s for reading\n", file_in);
284                     return 1;
285                 }
286             }
287
288             fout = stdout;
289             if (file_out)
290             {
```

```
291                 fout = fopen(file_out, "w");
292                 if (!fout)
293                 {
294                     fprintf(stderr, "Error opening file %s for writing\n", file_out);
295                     return 1;
296                 }
297             }
298 
299             // Redirection
300             dup2(fileno(fin), fileno(stdin));
301             dup2(fileno(fout), fileno(stdout));
302 
303             execvp(args[0], args);
304 
305             // Restore standard input and output
306             //dup2(fileno(stdin), fileno(fin));
307             //dup2(fileno(stdout), fileno(fout));
308 
309             dup2(STDERR_FILENO, STDOUT_FILENO);
310 
311             // Close file pointers if they were opened
312             if (file_in)
313                 fclose(fin);
314             if (file_out) {
315                 fclose(fout);
316             }
317 
318             printf("\nError, command not found: %s\n\n", args[0]);
319 
320             exit(-1);   // Si execvp no se ejecuta, es porque el comando es erroneo.
321         }
322 
323     } // end while
324 }
```