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

```c
/*-------------------------------------------------------
UNIX Shell Project
job_control module

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

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

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include "job_control.h"

// ------------------------------------------------------------------------
//  get_command() reads in the next command line, separating it into distinct tokens
//  using whitespace as delimiters. setup() sets the args parameter as a
//  null-terminated string.
// ------------------------------------------------------------------------

void get_command(char inputBuffer[], int size, char *args[],int *background)
{
    int length, /* # of characters in the command line */
        i,      /* loop index for accessing inputBuffer array */
        start,  /* index where beginning of next command parameter is */
        ct;     /* index of where to place the next parameter into args[] */

    ct = 0;
    *background=0;

    /* read what the user enters on the command line */
    length = read(STDIN_FILENO, inputBuffer, size);

    start = -1;
    if (length == 0)
    {
        printf("\nBye\n");
        exit(0);            /* ^d was entered, end of user command stream */
    }
    if (length < 0){
        perror("error reading the command");
        exit(-1);           /* terminate with error code of -1 */
    }

    /* examine every character in the inputBuffer */
    for (i=0;i<length;i++)
    {
        switch (inputBuffer[i])
        {
        case ' ':
        case '\t' :               /* argument separators */
            if(start != -1)
```

```
56              {
57                  args[ct] = &inputBuffer[start];    /* set up pointer */
58                  ct++;
59              }
60              inputBuffer[i] = '\0'; /* add a null char; make a C string */
61              start = -1;
62              break;
63
64          case '\n':                 /* should be the final char examined */
65              if (start != -1)
66              {
67                  args[ct] = &inputBuffer[start];
68                  ct++;
69              }
70              inputBuffer[i] = '\0';
71              args[ct] = NULL; /* no more arguments to this command */
72              break;
73
74          default :             /* some other character */
75
76              if (inputBuffer[i] == '&') // background indicator
77              {
78                  *background  = 1;
79                  if (start != -1)
80                  {
81                      args[ct] = &inputBuffer[start];
82                      ct++;
83                  }
84                  inputBuffer[i] = '\0';
85                  args[ct] = NULL; /* no more arguments to this command */
86                  i=length; // make sure the for loop ends now
87
88              }
89              else if (start == -1) start = i;  // start of new argument
90          }  // end switch
91      }  // end for
92      args[ct] = NULL; /* just in case the input line was > MAXLINE */
93  }
94
95
96  // -----------------------------------------------------------------------
97  /* devuelve puntero a un nodo con sus valores inicializados,
98  devuelve NULL si no pudo realizarse la reserva de memoria*/
99  job * new_job(pid_t pid, const char * command, enum job_state state)
100 {
101     job * aux;
102     aux=(job *) malloc(sizeof(job));
103     aux->pgid=pid;
104     aux->state=state;
105     aux->command=strdup(command);
106     aux->next=NULL;
107     return aux;
108 }
109
110 // -----------------------------------------------------------------------
111 /* inserta elemento en la cabeza de la lista */
112 void add_job (job * list, job * item)
113 {
114     job * aux=list->next;
115     list->next=item;
```

```c
    item->next=aux;
    list->pgid++;

}

// -----------------------------------------------------------------------
/* elimina el elemento indicado de la lista
devuelve 0 si no pudo realizarse con exito */
int delete_job(job * list, job * item)
{
    job * aux=list;
    while(aux->next!= NULL && aux->next!= item) aux=aux->next;
    if(aux->next)
    {
        aux->next=item->next;
        free(item->command);
        free(item);
        list->pgid--;
        return 1;
    }
    else
        return 0;

}
// -----------------------------------------------------------------------
/* busca y devuelve un elemento de la lista cuyo pid coincida con el indicado,
devuelve NULL si no lo encuentra */
job * get_item_bypid  (job * list, pid_t pid)
{
    job * aux=list;
    while(aux->next!= NULL && aux->next->pgid != pid) aux=aux->next;
    return aux->next;
}
// -----------------------------------------------------------------------
job * get_item_bypos( job * list, int n)
{
    job * aux=list;
    if(n<1 || n>list->pgid) return NULL;
    n--;
    while(aux->next!= NULL && n) { aux=aux->next; n--;}
    return aux->next;
}

// -----------------------------------------------------------------------
/*imprime una linea en el terminal con los datos del elemento: pid, nombre ... */
void print_item(job * item)
{

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

// -----------------------------------------------------------------------
/*recorre la lista y le aplica la funcion pintar a cada elemento */
void print_list(job * list, void (*print)(job *))
{
    int n=1;
    job * aux=list;
    printf("Contents of %s:\n",list->command);
    while(aux->next!= NULL)
```

```
175     {
176         printf(" [%d] ",n);
177         print(aux->next);
178         n++;
179         aux=aux->next;
180     }
181 }
182 
183 // -----------------------------------------------------------------------
184 /* interpretar valor status que devuelve wait */
185 enum status analyze_status(int status, int *info)
186 {
187     // el proceso se ha suspendido
188     if (WIFSTOPPED (status))
189     {
190         *info=WSTOPSIG(status);
191         return(SUSPENDED);
192     }
193     // el proceso se ha reanudado
194     else if (WIFCONTINUED(status))
195     {
196         *info=0;
197         return(CONTINUED);
198     }
199     else
200     {
201         // el proceso ha terminado
202         if (WIFSIGNALED (status))
203         {
204             *info=WTERMSIG (status);
205             return(SIGNALED);
206         }
207         else
208         {
209             *info=WEXITSTATUS(status);
210             return(EXITED);
211         }
212     }
213     return -1;
214 }
215 
216 // -----------------------------------------------------------------------
217 // cambia la accion de las seÃ±ales relacionadas con el terminal
218 void terminal_signals(void (*func) (int))
219 {
220     signal (SIGINT,  func); // crtl+c interrupt tecleado en el terminal
221     signal (SIGQUIT, func); // ctrl+\ quit tecleado en el terminal
222     signal (SIGTSTP, func); // crtl+z Stop tecleado en el terminal
223     signal (SIGTTIN, func); // proceso en segundo plano quiere leer del terminal
224     signal (SIGTTOU, func); // proceso en segundo plano quiere escribir en el terminal
225 }
226 
227 // -----------------------------------------------------------------------
228 void block_signal(int signal, int block)
229 {
230     /* declara e inicializa máscara */
231     sigset_t block_sigchld;
232     sigemptyset(&block_sigchld );
233     sigaddset(&block_sigchld,signal);
234     if(block)
```

```
235     {
236         /* bloquea señal */
237         sigprocmask(SIG_BLOCK, &block_sigchld, NULL);
238     }
239     else
240     {
241         /* desbloquea señal */
242         sigprocmask(SIG_UNBLOCK, &block_sigchld, NULL);
243     }
244 }
245
246
247
```