~\Desktop\ING DEL SOFTWARE\SOFTWARE 2_2\SISTEMAS OPERATIVOS\Prácticas\Práctica 4\prShellAmpliaciones\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 *respawnable)
{
    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;
    *respawnable=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 ' ':
```

```c
        case '\t' :               /* argument separators */
            if(start != -1)
            {
                args[ct] = &inputBuffer[start];    /* set up pointer */
                ct++;
            }
            inputBuffer[i] = '\0'; /* add a null char; make a C string */
            start = -1;
            break;

        case '\n':                 /* should be the final char examined */
            if (start != -1)
            {
                args[ct] = &inputBuffer[start];
                ct++;
            }
            inputBuffer[i] = '\0';
            args[ct] = NULL; /* no more arguments to this command */
            break;

        default :             /* some other character */

            if (inputBuffer[i] == '&') // background indicator
            {
                *background  = 1;
                if (start != -1)
                {
                    args[ct] = &inputBuffer[start];
                    ct++;
                }
                inputBuffer[i] = '\0';
                args[ct] = NULL; /* no more arguments to this command */
                i=length; // make sure the for loop ends now

            }
            else if (inputBuffer[i] == '+') // respawnable indicator
            {
                *respawnable  = 1;
                if (start != -1)
                {
                    args[ct] = &inputBuffer[start];
                    ct++;
                }
                inputBuffer[i] = '\0';
                args[ct] = NULL; /* no more arguments to this command */
                i=length; // make sure the for loop ends now

            }
            else if (start == -1) start = i;  // start of new argument
        }  // end switch
    }  // end for
    args[ct] = NULL; /* just in case the input line was > MAXLINE */
}


// -----------------------------------------------------------------------
/* devuelve puntero a un nodo con sus valores inicializados,
devuelve NULL si no pudo realizarse la reserva de memoria*/
job * new_job(pid_t pid, const char * command, enum job_state state)
{
```

```c
    job * aux;
    aux=(job *) malloc(sizeof(job));
    aux->pgid=pid;
    aux->state=state;
    aux->command=strdup(command);
    aux->next=NULL;
    return aux;
}

// -----------------------------------------------------------------------
/* inserta elemento en la cabeza de la lista */
void add_job (job * list, job * item)
{
    job * aux=list->next;
    list->next=item;
    item->next=aux;
    list->pgid++;

}

// -----------------------------------------------------------------------
/* inserta elemento respawnable en la cabeza de la lista */
void add_respawnable_job (job * list, job * item, char **args)  // Amp 1
{
    job * aux=list->next;
    item->args = (char**) malloc(sizeof(char)*200);
    list->next=item;
    item->next=aux;

    for (int i=0; args[i] != NULL; i++) {
        item->args[i] = strdup(args[i]);
    }

    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)
```

```c
{
    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)
    {
        printf(" [%d] ",n);
        print(aux->next);
        n++;
        aux=aux->next;
    }
}

// -----------------------------------------------------------------------
/*recorre la lista y le aplica la funcion pintar a cada elemento background */
void print_background_list(job * list, void (*print)(job *))    // Amp extra
{
    int n=1;
    job * aux=list;
    printf("Background contents of %s:\n",list->command);
    while(aux->next!= NULL)
    {
        if ((aux->next)->state == BACKGROUND) {
            printf(" [%d] ",n);
            print(aux->next);
        }

        n++;
        aux=aux->next;
    }
}

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

```c
/*recorre la lista y le aplica la funcion pintar a cada elemento stopped*/
void print_stopped_list(job * list, void (*print)(job *))   // Amp extra
{
    int n=1;
    job * aux=list;
    printf("Stopped contents of %s:\n",list->command);
    while(aux->next!= NULL)
    {
        if ((aux->next)->state == STOPPED) {
            printf(" [%d] ",n);
            print(aux->next);
        }

        n++;
        aux=aux->next;
    }
}

// -----------------------------------------------------------------------
/*recorre la lista y le aplica la funcion pintar a cada elemento respawnable*/
void print_respawnable_list(job * list, void (*print)(job *))   // Amp extra
{
    int n=1;
    job * aux=list;
    printf("Respawnable contents of %s:\n",list->command);
    while(aux->next!= NULL)
    {
        if ((aux->next)->state == RESPAWNABLE) {
            printf(" [%d] ",n);
            print(aux->next);
        }

        n++;
        aux=aux->next;
    }
}

// -----------------------------------------------------------------------
/* interpretar valor status que devuelve wait */
enum status analyze_status(int status, int *info)
{
    // el proceso se ha suspendido
    if (WIFSTOPPED (status))
    {
        *info=WSTOPSIG(status);
        return(SUSPENDED);
    }
    // el proceso se ha reanudado
    else if (WIFCONTINUED(status))
    {
        *info=0;
        return(CONTINUED);
    }
    else
    {
        // el proceso ha terminado
        if (WIFSIGNALED (status))
        {
            *info=WTERMSIG (status);
            return(SIGNALED);
```

```
294         }
295         else
296         {
297             *info=WEXITSTATUS(status);
298             return(EXITED);
299         }
300     }
301     return -1;
302 }
303 
304 // ---------------------------------------------------------------------
305 // cambia la accion de las seÃ±ales relacionadas con el terminal
306 void terminal_signals(void (*func) (int))
307 {
308     signal (SIGINT,  func); // crtl+c interrupt tecleado en el terminal
309     signal (SIGQUIT, func); // ctrl+\ quit tecleado en el terminal
310     signal (SIGTSTP, func); // crtl+z Stop tecleado en el terminal
311     signal (SIGTTIN, func); // proceso en segundo plano quiere leer del terminal
312     signal (SIGTTOU, func); // proceso en segundo plano quiere escribir en el terminal
313 }
314 
315 // ---------------------------------------------------------------------
316 void block_signal(int signal, int block)
317 {
318     /* declara e inicializa máscara */
319     sigset_t block_sigchld;
320     sigemptyset(&block_sigchld );
321     sigaddset(&block_sigchld,signal);
322     if(block)
323     {
324         /* bloquea señal */
325         sigprocmask(SIG_BLOCK, &block_sigchld, NULL);
326     }
327     else
328     {
329         /* desbloquea señal */
330         sigprocmask(SIG_UNBLOCK, &block_sigchld, NULL);
331     }
332 }
333 
334 
335 
```