```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);
 length = 0;
    switch (inputBuffer[0]) {
        case 4:
            length--;
            break;
        default:
            while (inputBuffer[length] != '\n') length++;
    }
    length++;

 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)
   {
```

```
    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;
    *respawnable = 0;

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

```
// Crea el historial y devuelve un puntero a un nodo con sus valores inicializados
history * new_history(){
 history *aux=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++;

}


void add_command(history ** hist, char **args, enum job_state est){
 history * item = (history *) malloc(sizeof(struct history_));

 item->command=strdup(args[0]);

 if(args!=NULL){
  item->args=(char**) malloc(128*sizeof(char*));

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

    item->state = est;
 item->next = NULL;


 if(*hist==NULL){
  *hist=item;
  (*hist)->prev=NULL;
  (*hist)->last=item;
 }else{
  (*hist)->last->next=item;
  item->prev= (*hist)->last;
  (*hist)->last=item;

 }
}

// ---------------------------------------------------------------------
/* inserta elemento respawnable en la cabeza de la lista */

void add_job_respawnable (job * list, job * item, char** args)
{
 int i=0;
 job * aux=list->next;
 item->args=(char**) malloc(sizeof(char*)*128);

 list->next=item;
 item->next=aux;
 while(args[i]!=NULL){
```

```
 item->args[i]=strdup(args[i]);
 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)
{
 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;
}

history* get_com_bypos(history* list, int n){
 history* aux=NULL;
 int encontrado=0;
 int j=0;
 while(list!=NULL && !encontrado){
  if(n == j){
   aux= list;
   encontrado==1;
  }
  j++;
  list=list->next;
 }
 return aux;

}




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

```c
/*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;
 }
}

void print_history(history * historial){
    history *history = historial;

    if (history == NULL) {
        printf("Empty history\n");
        fflush(stdout);
    } else {
        int j = 0;
        while (history != NULL) {
            printf("[%d] ", j);
            int i = 0;
            while((history->args)[i] != NULL){
                printf("%s ", (history->args)[i]);
                i++;
            }
            //añado '&' o '+'
            if (history->state==BACKGROUND) {
                printf("&");
            } else if (history->state==RESPAWNABLE) {
                printf("+");
            }

            printf("\n");
            fflush(stdout);

            history = history->next;
            j++;
        }
    }
}


// ---------------------------------------------------------------------
/* 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);
  }
  else
  {
   *info=WEXITSTATUS(status);
   return(EXITED);
  }
 }
 return -1;
}

// ---------------------------------------------------------------------
// cambia la accion de las seÃ±ales relacionadas con el terminal
void terminal_signals(void (*func) (int))
{
 signal (SIGINT,  func); // crtl+c interrupt tecleado en el terminal
 signal (SIGQUIT, func); // ctrl+\ quit tecleado en el terminal
 signal (SIGTSTP, func); // crtl+z Stop tecleado en el terminal
 signal (SIGTTIN, func); // proceso en segundo plano quiere leer del terminal
 signal (SIGTTOU, func); // proceso en segundo plano quiere escribir en el terminal
}

// ---------------------------------------------------------------------
void block_signal(int signal, int block)
{
 /* declara e inicializa máscara */
 sigset_t block_sigchld;
 sigemptyset(&block_sigchld );
 sigaddset(&block_sigchld,signal);
 if(block)
 {
  /* bloquea señal */
  sigprocmask(SIG_BLOCK, &block_sigchld, NULL);
 }
 else
 {
  /* desbloquea señal */
  sigprocmask(SIG_UNBLOCK, &block_sigchld, NULL);
 }
}
```