```c
/**
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)

**/

/*
 * Nombre: Galo Pérez Gallego
 * DNI: 79289212R
 */

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

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

#define RESET "\e[0;37m"
#define GREEN "\e[0;92m"
#define BLUE "\e[0;96m"
#define RED "\033[1;31m"
#define DARKBLUE "\x1b[34;1;1m"


job *lista_jobs;
history *historial;



void print_start(){
   char directorio[500];
   getcwd(directorio,sizeof(directorio));
   printf(GREEN"\ngalomax@DESKTOP-LUR32DQ:"RESET);
   printf(DARKBLUE"%s $ "RESET,directorio);
   fflush(stdout);
}

void manejadorSIGCHLD(int sig) {
 block_SIGCHLD();
 job_iterator iter = get_iterator(lista_jobs);
 int status, info;
 enum status status_res;
 pid_t pid_wait;
 job *the_job;
 pid_t pid_Resp;

 while(has_next(iter)) {
  the_job = next(iter);

  // Espero con WNOHANG, es decir, no me bloqueo
  // Con WUNTRACED sé si el grupo de procesos recibió un CTRL+Z
  // Con WCONTINUED sé si el grupo de procesos recibió un SIGCONT
  // Si el grupo de procesos por el que pregunto no salió ni recibió ninguna señal obtendré un 0 y pasaré a preguntar por el siguiente

  pid_wait = waitpid(the_job->pgid, &status, WNOHANG | WUNTRACED | WCONTINUED); //tengo que ponerle todas las opciones, y para combinarlas uso el or (|)
  // Hacer el waitpid sin opciones te bloquea (y no queremos bloquarnos para consultar), esto nos lo evita WNOHANG
  if (pid_wait == the_job->pgid) { //para comprobar que el waitpid ha acabado bien, lo comparo con el pgid del trabajo
   status_res = analyze_status(status, &info);
   //Veo en cual ha acabado: enum status { SUSPENDED, SIGNALED, EXITED, CONTINUED};
   if (status_res == SUSPENDED) { //suspendido, alguien busca pararlo
    printf(BLUE"Background job %s... pid: %d, command: %s\n"RESET, status_strings[status_res], the_job->pgid, the_job->command);
    // Actualizamos la lista de tareas
    the_job->state = STOPPED;

   } else if (status_res == EXITED) {
    // Si ha terminado, hay que quitarlo de la lista de tareas (a no ser que este en respawnable)
    printf(BLUE"Background job %s... pid: %d, command: %s\n"RESET, status_strings[status_res], the_job->pgid, the_job->command);
    if (the_job->state == RESPAWNABLE) { // si está en respawnable, se vuelve a lanzar una vez terminado
     printf(BLUE"Respawnable job running... pid: %d, command: %s\n"RESET, the_job->pgid, the_job->command);

     pid_Resp = fork(); // creamos nuevo proceso para relanzarlo

     if (pid_Resp < 0) {
      printf(RED"No se pudo crear el proceso\n"RESET);
     } else if (pid_Resp == 0) { // estamos en el hijo
      new_process_group(getpid()); // por si se planifica el hijo antes
      restore_terminal_signals();
      execvp(the_job->command, the_job->args); // puedo hacerlo por la modificación de estructura job
      printf(RED"Error, command not found\n"RESET);
      exit(-1);
     } else { // estamos en el padre
      new_process_group(pid_Resp); // por si se planifica el padre antes
      the_job->pgid = pid_Resp; // ha cambiado el pgid, actualizo la lista

     }

    } else {
     // Elimino la tarea de la lista
     delete_job(lista_jobs, the_job);
    }


   } else if (status_res == CONTINUED) {
    printf(BLUE"Background job %s... pid: %d, command: %s\n"RESET, status_strings[status_res], the_job->pgid, the_job->command);
    // Si una tarea en background recibe la señal de continuar, sigue en background
    the_job->state = BACKGROUND;

   } else if (status_res == SIGNALED) {
    printf(BLUE"Background job %s... pid: %d, command: %s\n"RESET, status_strings[status_res], the_job->pgid, the_job->command);
    // Cualquier otra señal que no sea STOPPED, CONTINUED, o EXITED, mata a la tarea
    // Elimino la tarea de la lista
    delete_job(lista_jobs, the_job);

   }
  }
 }
 unblock_SIGCHLD();
}

void JobsCommand() {

 if(empty_list(lista_jobs))
  printf("Empty job list");
 else
  print_job_list(lista_jobs);
}
```

```
void FgCommand(int pos) {

 int status, info;
 job *job = get_item_bypos(lista_jobs, pos);

 job -> state = FOREGROUND;
 printf(BLUE"%s pid: %d command: %s \n"RESET, state_strings[job->state], job->pgid, job->command);

 set_terminal(job -> pgid);
 killpg(job -> pgid, SIGCONT);
 waitpid(job -> pgid, &status, WUNTRACED); //WUNTRACED para tener en cuenta la suspensión de hijos

 set_terminal(getpid());

 //Comprobamos si se ha suspendido, sino, ha terminado y se elimina de la lista
 int status_job = analyze_status(status, &info);
 if (status_job == SUSPENDED) {
  job -> state = STOPPED;
 } else {
  delete_job(lista_jobs, job);
 }
}

void BgCommand(int pos) {

 job *job = get_item_bypos(lista_jobs, pos);

 if (job -> state == STOPPED || job->state == RESPAWNABLE) {

  job -> state = BACKGROUND;
  killpg(job -> pgid, SIGCONT);
 }
}

char getch()
{
 int shell_terminal = STDIN_FILENO;
 struct termios conf;
 struct termios conf_new;
 char c;
 tcgetattr(shell_terminal,&conf); /* leemos la configuracion actual */
 conf_new = conf;
 conf_new.c_lflag &= (~(ICANON|ECHO)); /* configuramos sin buffer ni eco */
 conf_new.c_cc[VTIME] = 0;
 conf_new.c_cc[VMIN] = 1;
 tcsetattr(shell_terminal,TCSANOW,&conf_new); /* establecer configuracion */
 c = getc(stdin); /* leemos el caracter */
 tcsetattr(shell_terminal,TCSANOW,&conf); /* restauramos la configuracion */
 return c;
}

void resetLine(char cmd[]) {
  int i = 0, max = strlen(cmd);
  for (i; i < max; i++) printf(" ");
}

void readInput(char inputBuffer[]) {
 /* Las teclas de cursor devuelven una secuencia de 3 caracteres, 27 - 91 -
(65, 66, 67 ó 68) */
 printf("\033[s");

 int first=0;

  history *pointer = NULL;
 if(historial!=NULL){
  pointer=historial->last;
```

```
}

char trasCursor[MAX_LINE+1];
trasCursor[0] = '\0';
int tamTrasCursor = 0;

int readCmd = 0, idBuff = 0, tamInput = 0, cont = 1;
char sec[3];

int i, j;
char c;

while (cont) {
 sec[0] = getch();
 switch (sec[0])
 {
 case 27:
  sec[1] = getch();
  if (sec[1] == 91) // 27,91,...
  {
   sec[2] = getch();
  switch (sec[2]){

   case 65: /* ARRIBA */
    if (historial != NULL &&  pointer->prev != NULL) {
     printf("\033[u");
     resetLine(inputBuffer);
     printf("\033[u");


     if(first){
      pointer = pointer->prev;
     }

     if(pointer==historial->last)
      first=1;



     strcpy(inputBuffer, pointer->command);

     int ar=1;
     int length= strlen(pointer->command);
     while(pointer->args[ar]!=NULL){
      strcat(inputBuffer," ");
      length++;
      strcat(inputBuffer, pointer->args[ar]);
      length+= strlen(pointer->args[ar]);
      ar++;
     }

     if(pointer->state==BACKGROUND){
      strcat(inputBuffer, " &");
      length+=2;
     }else if(pointer->state==RESPAWNABLE){
      strcat(inputBuffer, " +");
      length+=2;
     }

     idBuff = length;
     printf("%s", inputBuffer);
     trasCursor[0] = '\0';
     tamTrasCursor = 0;
    }
    break;
   case 66: /* ABAJO */
    /*if (historial != NULL) {
     if (pointer->next == historial->next) pointer = aux;
    else pointer = pointer->next;*/
    if(pointer!=NULL){
     printf("\033[u");
     resetLine(inputBuffer);
     printf("\033[u");
     idBuff = 0;
     if (pointer->next != NULL) {
      pointer=pointer->next;
      strcpy(inputBuffer, pointer->command);

      if(pointer == historial->last)
       first=0;


      int ar=1;
      int length= strlen(pointer->command);
      while(pointer->args[ar]!=NULL){
       strcat(inputBuffer," ");
       length++;
       strcat(inputBuffer, pointer->args[ar]);
       length+= strlen(pointer->args[ar]);
       ar++;
      }

      if(pointer->state==BACKGROUND){
       strcat(inputBuffer, " &");
       length+=2;
      }else if(pointer->state==RESPAWNABLE){
       strcat(inputBuffer, " +");
       length+=2;
      }

      idBuff = strlen(pointer->command);
      printf("%s", pointer->command);
     }
     trasCursor[0] = '\0';
     tamTrasCursor = 0;
    }
    break;
   case 67: /* DERECHA */
    if (tamTrasCursor != 0) {
     printf("\033[1C");
     idBuff++;
     i = 0;
     for (i; i < tamTrasCursor; i++) {
      trasCursor[i] = trasCursor[i+1];
     }
     tamTrasCursor--;
    }
    break;
   case 68: /* IZQUIERDA */
    if (idBuff != 0) {
     printf("\033[1D");
     idBuff--;
     c = inputBuffer[idBuff];
     i = tamTrasCursor;
     for (i; i >= 0; i--) {
      trasCursor[i+1] = trasCursor[i];
     }
     tamTrasCursor++;
     trasCursor[0] = c;
    }
    break;
   }
```

```
   }
   break;
   case 127: /* BORRAR */
    if (idBuff > 0) {
     printf("\033[1D%s \033[1D", trasCursor);
     i = tamTrasCursor;
     for (i; i > 0; i--) {
      printf("\033[1D");
     }
     i = idBuff;
     for (i; i < tamInput; i++) {
     inputBuffer[i-1] = inputBuffer[i];
     }
     idBuff--;
     tamInput--;
     i = 0;
     j = idBuff;
     while (i < tamTrasCursor) {
      inputBuffer[j] = trasCursor[i];
      i++;
      j++;
     }
   }
   break;
   case 4: /* ^D */
    inputBuffer[0] = sec[0];
    cont = 0;
    break;
   case 10: /* \n */
    cont = 0;
    i = idBuff;
    j = 0;
    for (i; i < idBuff+tamTrasCursor; i++) {
     inputBuffer[i] = trasCursor[j];
     j++;
    }
    idBuff = i;
    inputBuffer[idBuff] = '\n';
    inputBuffer[idBuff+1] = '\0';
    printf("\n");
    break;
   default: /* CUALQUIER OTRO CARACTER */
    tamInput++;
    printf("%c%s", sec[0], trasCursor);
    i = tamTrasCursor;
    for (i; i > 0; i--) {
     printf("\033[1D");
    }
    inputBuffer[idBuff] = sec[0];
    idBuff++;
   }
  }
 }

int esNumero(char linea[]) {
 int esNumero = 1, i = 0;
 while (esNumero == 1 && i < strlen(linea)) {
  if (isdigit(linea[i]) == 0) {
   esNumero = 0;
  }
  i++;
 }
 return esNumero;
}
```

```c
int piped( char** args){
 int f=0;
 int i=0;
 while(args[i]!=NULL && !f){

  if(!strcmp(args[i],"|"))
   f=1;
  i++;
 }
 return f;
}

// -----------------------------------------------------------------------
//                            MAIN
// -----------------------------------------------------------------------

int main(void)
{
 char inputBuffer[MAX_LINE]; /* buffer to hold the command entered */
 int background;             /* equals 1 if a command is followed by '&' */
 int respawnable;        /* equals 1 if a command is followed by '+' */
 char *args[MAX_LINE/2];     /* command line (of 256) has max of 128 arguments */
 char* first[MAX_LINE/2];
 char* second[MAX_LINE/2];
 // 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() */

 int time,timeout,pid_timeout;

 historial = new_history();

 ignore_terminal_signals();
 signal(SIGCHLD,manejadorSIGCHLD);

 lista_jobs=new_list("Background tasks");


 while (1)   /* Program terminates normally inside get_command() after ^D is typed*/
 {
  timeout=0;
  print_start();

  fflush(stdout);
  readInput(inputBuffer);

  get_command(inputBuffer, MAX_LINE, args, &background,&respawnable);  /* get next command */



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


  if(strcmp(args[0],"historial")){

   add_command(&historial,args,respawnable==1? RESPAWNABLE : background==1? BACKGROUND : FOREGROUND);

  }

  /* 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], "com") == 0){

   com:

   printf("Available commands:\n");
   printf("- com: Show available commands\n");
   printf("- cd [directorY]: change directory\n");
   printf("- jobs: shoe executed jobs\n");
   printf("- fg -> switches the first job running in the background into the foreground\n");
   printf("- fg [numJob]: switches the numJob job running in the background into the foreground\n");
   printf("- bg: places the first job in background\n");
   printf("- bg [numJob]: places the numJob job in background\n");
   printf("- time-out [Seg] [Com]: executes the command com during Seg seconds\n");
   printf("- historial [n]: shows the list of all commands used before, if historial is followed by a numbre, executes the n command of the list\n");
   printf("- pipe '|' : concat commands");

  }else if(!strcmp(args[0],"cd")){ //Comandos internos
    chdir(args[1]);

  }else if(!strcmp(args[0],"jobs")){
   JobsCommand();
  }else if(!strcmp(args[0],"fg")){
   fg:
   block_SIGCHLD();
   if(!empty_list(lista_jobs)){

    int i=0;
    while(args[i]!=NULL){
     i++;
    }
    if(i>2)
     printf(RED"Invalid arguments for fg\n"RESET);
    else{
     if(args[1]==NULL)
      FgCommand(1);
     else{
      if (atoi(args[1]) < 1 || atoi(args[1]) > list_size(lista_jobs))  // atoi pasa un string a un int
       printf(RED"No job available at that index\n"RESET);
      else
       FgCommand(atoi(args[1]));  //Usamos la función atoi que convierte string en int
      }
    }
   }else{
    printf(RED"No jobs available\n"RESET);
   }
   unblock_SIGCHLD();

  }else if (!strcmp(args[0], "bg")) {
   bg:
   block_SIGCHLD();

   if (!empty_list(lista_jobs)) {
    if (args[1] == NULL) // se ejecuta sin parametros, sobre la última tarea
     BgCommand(1);
    else { // se ejecuta con parámetros, sobre la tarea de posición indicada
     if (atoi(args[1]) < 1 || atoi(args[1]) > list_size(lista_jobs))  // atoi pasa un string a un int
      printf("No job available at that index\n");
     else
      BgCommand(atoi(args[1]));
    }
   }
   unblock_SIGCHLD();

  }else if (!strcmp(args[0], "historial")) {
   his:


   if(args[1]==NULL){
    if(historial!=NULL)
     print_history(historial);
    else
     printf("Empty history");
    add_command(&historial,args,respawnable==1 ? RESPAWNABLE : background==1 ? BACKGROUND : FOREGROUND);

    continue;
   }else{
    if(atoi(args[1])!=0 || !strcmp(args[1],"0")){
     int i= atoi(args[1]);

     if (get_com_bypos(historial,i)==NULL){
      printf(RED"Error: incorrect index"RESET);
     }else{
      history* aux = get_com_bypos(historial,i);
      background = aux->state == BACKGROUND ? 1 : 0;
      respawnable = aux->state == RESPAWNABLE ? 1 : 0;

      int i = 0;
      printf(BLUE"Running: ");
      fflush(stdout);
      while (aux->args[i] != 0){
       printf("%s ", aux->args[i]);
       args[i] = strdup(aux->args[i]);
       i++;
       fflush(stdout);
      }
      printf("\n"RESET);
      args[i] = NULL;

      if(piped(args)){
       goto p;
      }else if(!strcmp(args[0],"jobs"))
       JobsCommand();
      else if(!strcmp(args[0],"cd"))
       chdir(args[1]);
      else if(!strcmp(args[0],"com"))
       goto com;
      else if(!strcmp(args[0],"fg"))
       goto fg;
      else if(!strcmp(args[0],"bg"))
       goto bg;
      else if(!strcmp(args[0],"historial"))
       goto his;
      else
       goto main;

     }
    }
   }

  }else if (piped(args)) {
   p:


   if (!strcmp(args[0],"|")) {
```

```
 printf("Pipe syntax error\n");
 continue;
} else {
 first[0] = strdup(args[0]);

}

int i = 1;
while (strcmp(args[i],"|")) {
 first[i] = strdup(args[i]);
 i++;
}
first[i] = NULL;

i++;

int j = 0;
if (args[i] == NULL) {
 printf("Pipe syntax error\n");
 continue;
} else {
 second[j] = strdup(args[i]);
}

i++; j++;
while (args[i] != NULL) {
 second[j] = strdup(args[i]);
 i++; j++;
}
second[j] = NULL;

 pid_fork = fork();
 if (pid_fork == 0) { // es el hijo padre
  int descf[2], fno;
  pipe(descf); /* se crea un pipe */

  int pid_fork2= fork();

  if (pid_fork2!=0){ // este es el hijo padre
   /* el proceso padre ejecuta el primer programa y cambia su
   salida estandar al pipe cerrando la entrada del pipe */
   fno=fileno(stdout);
   dup2(descf[1],fno);

   close(descf[0]);
   execvp(first[0],first);

  } else { // este es el hijo hijo
   /* el proceso hijo tiene una copia del pipe del padre,
   en el fork, ejecuta el segundo programa y cambia su
   entrada estandar por el pipe cerrando la salida del pipe */
   fno=fileno(stdin);
   dup2(descf[0],fno);

   close(descf[1]);
   execvp(second[0],second);

  }
 }
            continue;


     } else { //Comandos externos

main:
 if(!strcmp(args[0],"time-out")){

 if(args[1]!=NULL && args[2]!=NULL){
  timeout=1;
  time=atoi(args[1]);
  int i=0;
  while(args[i+2]!=NULL){
   args[i] = strdup(args[i+2]);
   args[i+2]=NULL;
   i++;
  }
  args[i]=NULL;

  }else{
   printf(RED"Invalid amount of arguments"RESET);
   continue;
  }
 }



 pid_fork = fork();

 if(pid_fork<0){ //Fallo en fork
  printf(RED"Procces could not be created"RESET);
  continue;
 }else if(pid_fork){ //Estamos en el padre (Shell)
  new_process_group(pid_fork);

  if(timeout){
   pid_timeout=fork();
   if(pid_timeout==0){
    sleep(time);
    killpg(pid_fork,SIGKILL);
    exit(0);
   }
  }


  if(background){
   printf(BLUE"Background job running... pid: %d, command: %s\n"RESET,pid_fork,args[0]);

   block_SIGCHLD();
   add_job(lista_jobs,new_job(pid_fork,args[0],background));
   unblock_SIGCHLD();
  }else if(respawnable && !(!strcmp(args[0],"sleep") && atoi(args[1])==0) ){ //Excluimos el caso en el que no le pasamos un número a la función sleep
   printf(BLUE"Respawnable job running... pid: %d, command: %s\n"RESET,pid_fork,args[0]);
   block_SIGCHLD();
   add_job_respawnable(lista_jobs,new_job(pid_fork,args[0],RESPAWNABLE),args);
   unblock_SIGCHLD();
  }else{ //foregound

   set_terminal(pid_fork);
   pid_wait = waitpid(pid_fork,&status,WUNTRACED);
   status_res = analyze_status(status, &info);

   if(pid_wait == pid_fork){

    // Compruebo si se ha hecho ctrl+Z
    if(status_res == SUSPENDED){
     block_SIGCHLD();
     add_job(lista_jobs,new_job(pid_fork,args[0],STOPPED));
     unblock_SIGCHLD();
    }
```

```
      printf(BLUE"Foreground pid: %d, command: %s, %s, info: %d\n"RESET, pid_wait,args[0],status_strings[status_res],info);
     }



     set_terminal(getpid());
    }

   }else{ //pid_fork=0 (estamos en el hijo)

   // Creo un nuevo process group id. Si el padre se planificó antes, lo hizo por mí
    new_process_group(getpid());
    if(!background && !respawnable){
     set_terminal(getpid());
    }
    // Activo las señales para el proceso hijo una vez asignado el terminal (tras el exec se heredan activas)
    restore_terminal_signals();
    execvp(args[0],args);
    printf(RED"Error: command not found '%s'\n"RESET, args[0]);
    fflush(stdout);
    exit(1);

   }
  }

 } // end while
}
```