

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

```

1  /*-----
2  UNIX Shell Project
3  job_control module
4
5  Sistemas Operativos
6  Grados I. Informatica, Computadores & Software
7  Dept. Arquitectura de Computadores - UMA
8
9  Some code adapted from "Fundamentos de Sistemas Operativos", Silberschatz et al.
10 -----*/
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <string.h>
15 #include <malloc.h>
16 #include "job_control.h"
17
18 // -----
19 //  get_command() reads in the next command line, separating it into distinct tokens
20 //  using whitespace as delimiters. setup() sets the args parameter as a
21 //  null-terminated string.
22 // -----
23
24 void get_command(char inputBuffer[], int size, char *args[],int *background, int
    *respawnable)
25 {
26     int length, /* # of characters in the command line */
27         i,      /* loop index for accessing inputBuffer array */
28         start,  /* index where beginning of next command parameter is */
29         ct;     /* index of where to place the next parameter into args[] */
30
31     ct = 0;
32     *background=0;
33     *respawnable=0;
34
35     /* read what the user enters on the command line */
36     length = read(STDIN_FILENO, inputBuffer, size);
37
38     start = -1;
39     if (length == 0)
40     {
41         printf("\nBye\n");
42         exit(0); /* ^d was entered, end of user command stream */
43     }
44     if (length < 0){
45         perror("error reading the command");
46         exit(-1); /* terminate with error code of -1 */
47     }
48
49     /* examine every character in the inputBuffer */
50     for (i=0;i<length;i++)
51     {
52         switch (inputBuffer[i])
53         {
54             case ' ':

```

```

55     case '\t' :                /* argument separators */
56         if(start != -1)
57         {
58             args[ct] = &inputBuffer[start];    /* set up pointer */
59             ct++;
60         }
61         inputBuffer[i] = '\0'; /* add a null char; make a C string */
62         start = -1;
63         break;
64
65     case '\n':                /* should be the final char examined */
66         if (start != -1)
67         {
68             args[ct] = &inputBuffer[start];
69             ct++;
70         }
71         inputBuffer[i] = '\0';
72         args[ct] = NULL; /* no more arguments to this command */
73         break;
74
75     default :                /* some other character */
76
77         if (inputBuffer[i] == '&') // background indicator
78         {
79             *background = 1;
80             if (start != -1)
81             {
82                 args[ct] = &inputBuffer[start];
83                 ct++;
84             }
85             inputBuffer[i] = '\0';
86             args[ct] = NULL; /* no more arguments to this command */
87             i=length; // make sure the for loop ends now
88
89         }
90         else if (inputBuffer[i] == '+') // respawnable indicator
91         {
92             *respawnable = 1;
93             if (start != -1)
94             {
95                 args[ct] = &inputBuffer[start];
96                 ct++;
97             }
98             inputBuffer[i] = '\0';
99             args[ct] = NULL; /* no more arguments to this command */
100             i=length; // make sure the for loop ends now
101
102         }
103         else if (start == -1) start = i; // start of new argument
104     } // end switch
105 } // end for
106 args[ct] = NULL; /* just in case the input line was > MAXLINE */
107 }
108
109
110 // -----
111 /* devuelve puntero a un nodo con sus valores inicializados,
112 devuelve NULL si no pudo realizarse la reserva de memoria*/
113 job * new_job(pid_t pid, const char * command, enum job_state state)
114 {

```

```

115     job * aux;
116     aux=(job *) malloc(sizeof(job));
117     aux->pgid=pid;
118     aux->state=state;
119     aux->command=strdup(command);
120     aux->next=NULL;
121     return aux;
122 }
123
124 // -----
125 /* inserta elemento en la cabeza de la lista */
126 void add_job (job * list, job * item)
127 {
128     job * aux=list->next;
129     list->next=item;
130     item->next=aux;
131     list->pgid++;
132 }
133
134 // -----
135 /* inserta elemento respawnable en la cabeza de la lista */
136 void add_respawnable_job (job * list, job * item, char **args) // Amp 1
137 {
138     job * aux=list->next;
139     item->args = (char**) malloc(sizeof(char)*200);
140     list->next=item;
141     item->next=aux;
142
143     for (int i=0; args[i] != NULL; i++) {
144         item->args[i] = strdup(args[i]);
145     }
146
147     list->pgid++;
148 }
149
150 // -----
151 /* elimina el elemento indicado de la lista
152 devuelve 0 si no pudo realizarse con exito */
153 int delete_job(job * list, job * item)
154 {
155     job * aux=list;
156     while(aux->next!= NULL && aux->next!= item) aux=aux->next;
157     if(aux->next)
158     {
159         aux->next=item->next;
160         free(item->command);
161         free(item);
162         list->pgid--;
163         return 1;
164     }
165     else
166         return 0;
167 }
168
169 // -----
170 /* busca y devuelve un elemento de la lista cuyo pid coincida con el indicado,
171 devuelve NULL si no lo encuentra */
172 job * get_item_bypid (job * list, pid_t pid)

```

```

175 {
176     job * aux=list;
177     while(aux->next!= NULL && aux->next->pgid != pid) aux=aux->next;
178     return aux->next;
179 }
180 // -----
181 job * get_item_bypos( job * list, int n)
182 {
183     job * aux=list;
184     if(n<1 || n>list->pgid) return NULL;
185     n--;
186     while(aux->next!= NULL && n) { aux=aux->next; n--;}
187     return aux->next;
188 }
189
190 // -----
191 /*imprime una linea en el terminal con los datos del elemento: pid, nombre ... */
192 void print_item(job * item)
193 {
194
195     printf("pid: %d, command: %s, state: %s\n", item->pgid, item->command,
196 state_strings[item->state]);
197 }
198 // -----
199 /*recorre la lista y le aplica la funcion pintar a cada elemento */
200 void print_list(job * list, void (*print)(job *))
201 {
202     int n=1;
203     job * aux=list;
204     printf("Contents of %s:\n",list->command);
205     while(aux->next!= NULL)
206     {
207         printf(" [%d] ",n);
208         print(aux->next);
209         n++;
210         aux=aux->next;
211     }
212 }
213
214 // -----
215 /*recorre la lista y le aplica la funcion pintar a cada elemento background */
216 void print_background_list(job * list, void (*print)(job *)) // Amp extra
217 {
218     int n=1;
219     job * aux=list;
220     printf("Background contents of %s:\n",list->command);
221     while(aux->next!= NULL)
222     {
223         if ((aux->next)->state == BACKGROUND) {
224             printf(" [%d] ",n);
225             print(aux->next);
226         }
227
228         n++;
229         aux=aux->next;
230     }
231 }
232
233 // -----

```

```
234 /*recorre la lista y le aplica la funcion pintar a cada elemento stopped*/
235 void print_stopped_list(job * list, void (*print)(job *)) // Amp extra
236 {
237     int n=1;
238     job * aux=list;
239     printf("Stopped contents of %s:\n",list->command);
240     while(aux->next!= NULL)
241     {
242         if ((aux->next)->state == STOPPED) {
243             printf(" [%d] ",n);
244             print(aux->next);
245         }
246         n++;
247         aux=aux->next;
248     }
249 }
250
251
252 // -----
253 /*recorre la lista y le aplica la funcion pintar a cada elemento respawnable*/
254 void print_respawnable_list(job * list, void (*print)(job *)) // Amp extra
255 {
256     int n=1;
257     job * aux=list;
258     printf("Respawnable contents of %s:\n",list->command);
259     while(aux->next!= NULL)
260     {
261         if ((aux->next)->state == RESPAWNABLE) {
262             printf(" [%d] ",n);
263             print(aux->next);
264         }
265         n++;
266         aux=aux->next;
267     }
268 }
269
270
271 // -----
272 /* interpretar valor status que devuelve wait */
273 enum status analyze_status(int status, int *info)
274 {
275     // el proceso se ha suspendido
276     if (WIFSTOPPED (status))
277     {
278         *info=WSTOPSIG(status);
279         return(SUSPENDED);
280     }
281     // el proceso se ha reanudado
282     else if (WIFCONTINUED(status))
283     {
284         *info=0;
285         return(CONTINUED);
286     }
287     else
288     {
289         // el proceso ha terminado
290         if (WIFSIGNALED (status))
291         {
292             *info=WTERMSIG (status);
293             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 señales relacionadas con el terminal
306 void terminal_signals(void (*func) (int))
307 {
308     signal (SIGINT,  func); // ctrl+c interrupt tecleado en el terminal
309     signal (SIGQUIT, func); // ctrl+\ quit tecleado en el terminal
310     signal (SIGTSTP, func); // ctrl+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
```