

Prueba de Laboratorio

Modelo A02 - Gestión de Procesos

APELLIDOS: _____

NOMBRE: _____

GRUPO DE LABORATORIO: _____

Indicaciones:

- No se permiten libros, apuntes ni teléfonos móviles sobre la mesa.
- Cuando tenga una solución (compilación + ejecución) al ejercicio muéstrela al profesor.
- Debe anotar su solución por escrito en el espacio disponible en este cuestionario.

Calificación

Enunciado

Construya, utilizando ANSI C estándar, un sistema compuesto por **tres ejecutables** que simule el funcionamiento que se detalla a continuación. El sistema contará con tres tipos de procesos: i) *manager*, ii) *PROCESADOR* y iii) *CONTADOR*.

El **proceso *manager*** será responsable de crear un número determinado de procesos de tipo *PROCESADOR* y de tipo *CONTADOR*, gestionando de manera adecuada su finalización y liberando los recursos previamente reservados. Este proceso abrirá un fichero, cuyo nombre recibirá por la línea de órdenes, y leerá su contenido línea a línea. Por cada línea creará un proceso *PROCESADOR* y uno *CONTADOR*.

Por una parte, los **procesos de tipo *PROCESADOR*** recibirán en el momento de su creación un número de línea, una línea y un patrón. Su función consistirá en comprobar si este patrón se corresponde con alguna de las palabras que conforman la línea recibida.

Por otra parte, los **procesos de tipo *CONTADOR*** recibirán en el momento de su creación un número de línea y una línea. Su función consistirá en contar el número de palabras que conforman la línea recibida.

La ruta al archivo a procesar y el patrón a buscar serán indicados por el usuario a través de la línea de órdenes al ejecutar el único proceso de tipo *manager*:

```
./exec/manager <archivo> <patrón>
```

donde <archivo> representa dicha ruta y <patrón> el propio patrón.

La **finalización de la simulación** tendrá lugar si se cumple una de las dos condiciones siguientes:

1. Todos los procesos de tipo *PROCESADOR* y *CONTADOR* finalizan su ejecución. El proceso *manager*, tras detectar esta situación, liberará recursos.
2. El usuario pulsa la combinación de teclas Ctrl + C. El proceso *manager*, tras detectar este evento, enviará una señal de finalización a todos los procesos de tipo *PROCESADOR* y *CONTADOR* que estén en ejecución y liberará recursos.

Resolución

Utilice el código fuente suministrado a continuación como plantilla para resolver el ejercicio. Este código no debe ser modificado. Únicamente debe incorporar su código en la sección indicada.

Ejemplo de ejecución

Una vez resuelto el ejercicio, si ejecuta el manager con los siguientes argumentos (make test),

```
./exec/manager data/test.txt What
```

el resultado debe ser similar al siguiente (cambiará el PID de los procesos PROCESADOR y CONTADOR, el orden de impresión y los valores generados de manera aleatoria):

```
[COUNTER 3031] The line '2' has 6 words
[PATTERN 3034] Pattern 'What' found in line 4
[COUNTER 3027] The line '0' has 4 words
[MANAGER] 24 processes created.
[PATTERN 3046] Pattern 'What' found in line 10
[PATTERN 3030] Pattern 'What' found in line 2
[COUNTER 3045] The line '9' has 3 words
[COUNTER 3037] The line '5' has 3 words
[COUNTER 3047] The line '10' has 4 words
[COUNTER 3039] The line '6' has 15 words
[COUNTER 3049] The line '11' has 17 words
[COUNTER 3033] The line '3' has 8 words
[COUNTER 3041] The line '7' has 10 words
[PATTERN 3036] Pattern 'What' found in line 5
[COUNTER 3043] The line '8' has 8 words
[COUNTER 3035] The line '4' has 5 words
[COUNTER 3029] The line '1' has 6 words

[MANAGER] Program termination (all the processes terminated).
```

✂ Anote la parte de finalización de la salida de la simulación con la siguiente lista de argumentos (make solution) respetando el formato de impresión del ejemplo de ejecución anterior:

```
./exec/manager data/solution.txt tortoise
```

Resultado:

Esqueleto de Código Fuente

A continuación se muestra el esqueleto de código fuente para resolver el ejercicio. Sólo debe incluir el código que completa los recuadros en blanco.

Makefile

```

1  DIROBJ := obj/
2  DIREXE := exec/
3  DIRHEA := include/
4  DIRSRC := src/
5
6  CFLAGS := -I$(DIRHEA) -c -Wall -ansi
7  LDLIBS := -lpthread -lrt
8  CC := gcc
9
10 all : dirs manager pattern counter
11
12 dirs:
13     mkdir -p $(DIROBJ) $(DIREXE)
14
15 manager: $(DIROBJ)manager.o
16     $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
17 pattern: $(DIROBJ)pattern.o
18     $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
19 counter: $(DIROBJ)counter.o
20     $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
21
22 $(DIROBJ)%.o: $(DIRSRC)%.c
23     $(CC) $(CFLAGS) $^ -o $@
24
25 test:
26     ./$(DIREXE)manager data/test.txt What
27 solution:
28     ./$(DIREXE)manager data/solution.txt tortoise
29
30 clean :
31     rm -rf *~ core $(DIROBJ) $(DIREXE) $(DIRHEA)*~ $(DIRSRC)*~

```

definitions.h

```

32 #define PATTERN_CLASS "PATTERN"
33 #define PATTERN_PATH "./exec/pattern"
34 #define COUNTER_CLASS "COUNTER"
35 #define COUNTER_PATH "./exec/counter"
36
37 /* Process class */
38 enum ProcessClass_t {PATTERN, COUNTER};
39
40 /* Process info */
41 struct TProcess_t {
42     enum ProcessClass_t class; /* PATTERN or COUNTER */
43     pid_t pid; /* Process ID */
44     char *str_process_class; /* String representation of the process class */
45 };

```

manager.c

```

46 #define _POSIX_SOURCE
47
48 #include <errno.h>
49 #include <linux/limits.h>
50 #include <signal.h>
51 #include <stdio.h>
52 #include <stdlib.h>
53 #include <string.h>
54 #include <sys/wait.h>
55 #include <sys/types.h>
56 #include <unistd.h>
57
58 #include <definitions.h>
59
60 /* Total number of processes */
61 int g_nProcesses;
62 /* 'Process table' (child processes) */
63 struct TProcess_t *g_process_table;
64
65 /* Process management */
66 void create_processes(const char *filename, const char *pattern);
67 void create_processes_by_class(enum ProcessClass_t class, int n_new_processes,
68                               int index_process_table, const char *line,
69                               const char *line_number_str, const char *pattern);
68 pid_t create_single_process(const char *path, const char *str_process_class,
69                             const char *line, const char *line_number_str,
70                             const char *pattern);
69 void get_str_process_info(enum ProcessClass_t class, char **path, char **str_process_class);
70 void init_process_table(int n_processes_pattern, int n_processes_counter);
71 void terminate_processes(void);
72 void wait_processes();


```

```

73  /* Auxiliar functions */
74  void free_resources();
75  void install_signal_handler();
76  void parse_argv(int argc, char *argv[], char **filename, char **pattern, int *lines);
77  void signal_handler(int signo);
78
79  /***** Main function *****/
80
81  int main(int argc, char *argv[]) {
82      char *filename = NULL, *pattern = NULL;
83      int lines = 0;
84
85      parse_argv(argc, argv, &filename, &pattern, &lines);
86      install_signal_handler();
87
88      init_process_table(lines, lines);
89      create_processes(filename, pattern);
90      wait_processes();
91
92      printf("\n[MANAGER] Program termination (all the processes terminated).\n");
93      free_resources();
94
95      return EXIT_SUCCESS;
96  }
97
98  /***** Process management *****/
99
100 void create_processes(const char *filename, const char *pattern) {
101     FILE *fp;
102     char line[PATH_MAX], line_number_str[3];
103     int line_number = 0;
104
105     if ((fp = fopen(filename, "r")) == NULL) {
106         fprintf(stderr, "Error opening file %s\n", filename);
107         exit(EXIT_FAILURE);
108     }
109
110     while (fgets(line, sizeof(line), fp) != NULL) {
111         sprintf(line_number_str, "%d", line_number);
112         create_processes_by_class(PATTERN, 1, line_number * 2, line, line_number_str, pattern);
113         create_processes_by_class(COUNTER, 1, line_number * 2 + 1, line, line_number_str, NULL);
114         line_number++;
115     }
116
117     printf("[MANAGER] %d processes created.\n", line_number * 2);
118     sleep(1);
119
120     fclose(fp);
121 }
122
123 void create_processes_by_class(enum ProcessClass t class, int n new processes,
124                               int index_process_table, const char *line,
125                               const char *line_number_str, const char *pattern) {
126     char *path = NULL, *str_process_class = NULL;
127     int i;
128     pid_t pid;
129
130     get_str_process_info(class, &path, &str_process_class);
131
132     for (i = index_process_table; i < (index_process_table + n new processes); i++) {
133         pid = create_single_process(path, str_process_class, line, line_number_str, pattern);
134         g_process_table[i].class = class;
135         g_process_table[i].pid = pid;
136         g_process_table[i].str_process_class = str_process_class;
137     }
138 }
139 pid_t create_single_process(const char *path, const char *str_process_class,
140                             const char *line, const char *line_number_str,
141                             const char *pattern) {
142     pid_t pid;
143
144     switch (pid = fork()) {
145     case -1:
146         fprintf(stderr, "[MANAGER] Error creating %s process: %s.\n",
147                 str_process_class, strerror(errno));
148         terminate_processes();
149         free_resources();
150         exit(EXIT_FAILURE);
151         /* Child process */
152     case 0:
153         if (execl(path, str_process_class, line, line_number_str, pattern, NULL) == -1) {
154             fprintf(stderr, "[MANAGER] Error using execl() in %s process: %s.\n",
155                     str_process_class, strerror(errno));
156             exit(EXIT_FAILURE);
157         }
158     }
159
160     /* Parent process */
161     return pid;
162 }

```

```
159 void get_str_process_info(enum ProcessClass t_class, char **path,  
                             char **str_process_class) {  
160     switch (class) {  
161     case PATTERN:  
162         *path = PATTERN_PATH;  
163         *str_process_class = PATTERN_CLASS;  
164         break;  
165     case COUNTER:  
166         *path = COUNTER_PATH;  
167         *str_process_class = COUNTER_CLASS;  
168         break;  
169     }  
170 }  
171  
172 void init_process_table(int n_processes_pattern, int n_processes_counter) {
```

 Incluya aquí el código para inicializar la tabla de procesos (Aprox. \approx 6 Líneas de código)

```
173 }  
174  
175 void terminate_processes(void) {
```

 Incluya aquí el código para finalizar procesos hijo (Aprox. \approx 10 Líneas de código)

```
176 }
```

```
177 void wait_processes() {
```

 Incluya aquí el código de espera a procesos hijo (*Aprox. ≈ 13 Líneas de código*)

```
178 }
179
180 /***** Auxiliar functions *****/
181
182 void free_resources() {
183     /* Free the 'process table' memory */
184     free(g_process_table);
185 }
186
187 void install_signal_handler() {
188     if (signal(SIGINT, signal_handler) == SIG_ERR) {
189         fprintf(stderr, "[MANAGER] Error installing signal handler: %s.\n",
190             strerror(errno));
191         exit(EXIT_FAILURE);
192     }
193 }
194
195 void parse_argv(int argc, char *argv[], char **filename, char **pattern, int *lines) {
196     FILE *fp;
197     int ch;
198
199     if (argc != 3) {
200         fprintf(stderr, "Error. Use: ./exec/manager <file> <pattern>.\n");
201         exit(EXIT_FAILURE);
202     }
203
204     *filename = argv[1];
205     *pattern = argv[2];
206
207     if ((fp = fopen(*filename, "r")) == NULL) {
208         fprintf(stderr, "Error opening file %s\n", *filename);
209         exit(EXIT_FAILURE);
210     }
211
212     while ((ch = fgetc(fp)) != EOF) {
213         if (ch == '\n') {
214             ++*lines;
215         }
216     }
217
218     fclose(fp);
219 }
220
221 void signal_handler(int signo) {
222     printf("\n[MANAGER] Program termination (Ctrl + C).\n");
223     terminate_processes();
224     free_resources();
225     exit(EXIT_SUCCESS);
226 }
```

pattern.c

```

226 #include <errno.h>
227 #include <signal.h>
228 #include <stdio.h>
229 #include <stdlib.h>
230 #include <string.h>
231 #include <unistd.h>
232
233 /* Program logic */
234 void run(char *line, int line_number, const char *pattern);
235
236 /* Auxiliar functions */
237 void install_signal_handler();
238 void parse_argv(int argc, char *argv[], char **line, int *line_number, char **pattern);
239 void signal_handler(int signo);
240
241 /***** Main function *****/
242
243 int main (int argc, char *argv[]) {
244     char *line = NULL, *pattern = NULL;
245     int line_number;
246
247     install_signal_handler();
248     parse_argv(argc, argv, &line, &line_number, &pattern);
249
250     run(line, line_number, pattern);
251
252     return EXIT_SUCCESS;
253 }
254
255 /***** Program logic *****/
256
257 void run(char *line, int line_number, const char *pattern) {
258     char *token;
259
260     token = strtok(line, " ");
261     while (token != NULL) {
262         if (strcmp(token, pattern) == 0) {
263             printf("[PATTERN %d] Pattern '%s' found in line %d\n",
264                 getpid(), pattern, line_number);
265             token = strtok(NULL, " ");
266         }
267     }
268 }
269
270 /***** Auxiliar functions *****/
271
272 void install_signal_handler() {
273     if (signal(SIGINT, signal_handler) == SIG_ERR) {
274         fprintf(stderr, "[PA %d] Error installing handler: %s.\n",
275             getpid(), strerror(errno));
276         exit(EXIT_FAILURE);
277     }
278 }
279
280 void parse_argv(int argc, char *argv[], char **line, int *line_number, char **pattern) {
281     if (argc != 4) {
282         fprintf(stderr, "[PATTERN %d] Error in the command line.\n", getpid());
283         exit(EXIT_FAILURE);
284     }
285
286     *line = argv[1];
287     *line_number = atoi(argv[2]);
288     *pattern = argv[3];
289 }
290
291 void signal_handler(int signo) {
292     printf("[PATTERN %d] terminated (SIGINT).\n", getpid());
293     exit(EXIT_SUCCESS);
294 }

```

counter.c

```

293 #include <errno.h>
294 #include <signal.h>
295 #include <stdio.h>
296 #include <stdlib.h>
297 #include <string.h>
298 #include <unistd.h>
299
300 /* Program logic */
301 void run(char *line, int line_number);
302
303 /* Auxiliar functions */
304 void install_signal_handler();
305 void parse_argv(int argc, char *argv[], char **line, int *line_number);
306 void signal_handler(int signo);
307
308 /***** Main function *****/
309
310 int main (int argc, char *argv[]) {
311     char *line = NULL;
312     int line_number;
313
314     install_signal_handler();
315     parse_argv(argc, argv, &line, &line_number);
316
317     run(line, line_number);
318
319     return EXIT_SUCCESS;
320 }
321
322 /***** Program logic *****/
323
324 void run(char *line, int line_number) {
325     int n_words = 0, inside_word = 0;
326     const char* it = line;
327
328     do {
329         switch (*it) {
330             case '\0':
331             case ' ': case '\t': case '\n': case '\r':
332                 if (inside_word) {
333                     inside_word = 0;
334                     n_words++;
335                 }
336                 break;
337             default:
338                 inside_word = 1;
339         }
340         while(*it++);
341     } while(*it);
342
343     printf("[COUNTER %d] The line '%d' has %d words\n", getpid(), line_number, n_words);
344 }
345 /***** Auxiliar functions *****/
346
347 void install_signal_handler() {
348     if (signal(SIGINT, signal_handler) == SIG_ERR) {
349         fprintf(stderr, "[PA %d] Error installing handler: %s.\n",
350             getpid(), strerror(errno));
351         exit(EXIT_FAILURE);
352     }
353 }
354
355 void parse_argv(int argc, char *argv[], char **line, int *line_number) {
356     if (argc != 3) {
357         fprintf(stderr, "[COUNTER %d] Error in the command line.\n", getpid());
358         exit(EXIT_FAILURE);
359     }
360     *line = argv[1];
361     *line_number = atoi(argv[2]);
362 }
363
364 void signal_handler(int signo) {
365     printf("[COUNTER %d] terminated (SIGINT).\n", getpid());
366     exit(EXIT_SUCCESS);
367 }

```