

Prueba de Laboratorio

Modelo A01 - Gestión de Procesos

APELLIDOS: _____

NOMBRE: _____

GRUPO DE LABORATORIO: _____

Indicaciones:

- No se permiten libros, apuntes ni teléfonos móviles sobre la mesa.
- Cuando tenga una solución al ejercicio muéstrela al profesor (compilación + ejecución).
- 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) *PA* y iii) *PB*.

El **proceso *manager*** será el responsable de crear un número determinado de procesos de tipo *PA* y de tipo *PB*, gestionando de manera adecuada su finalización y liberando los recursos previamente reservados.

Por una parte, los **procesos de tipo *PA*** simplemente *dormirán* un número aleatorio de segundos, definido entre 1 y el número indicado a través del primer argumento recibido por línea de órdenes. A continuación, finalizarán su ejecución.

Por otra parte, los **procesos de tipo *PB*** ejecutarán un bucle infinito en el que en cada iteración *dormirán* un número aleatorio de segundos, definido entre 1 y el número indicado en el primer argumento recibido por línea de órdenes.

El número de procesos de cada tipo, así como el tiempo máximo de espera, será indicado por el usuario a través de la línea de órdenes al ejecutar el único proceso de tipo *manager*:

```
./exec/manager <n_procesos_PA> <n_procesos_PB> <t_max_espera>
```

donde `<n_procesos_PA>` representa el número de procesos de tipo *PA* a crear, `<n_procesos_PB>` el número de procesos de tipo *PB* y, finalmente, `<t_max_espera>` representa el tiempo máximo de espera de los procesos (tiempo que *duermen*) que será comunicado a los procesos *PA* y *PB* en el momento de su creación.

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

1. Todos los procesos de tipo *PA* finalizan su ejecución. El proceso *manager*, tras detectar esta situación, enviará una señal de finalización a los procesos de tipo *PB* y 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 *PA* y *PB* 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 3 2 5
```

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

```
[MANAGER] 2 PB processes created.
[PB 3108] sleeps 2 seconds.
[PB 3109] sleeps 1 seconds.
[MANAGER] 3 PA processes created.
[PB 3109] sleeps 1 seconds.
[PA 3111] sleeps 2 seconds.
[PA 3112] sleeps 5 seconds.
[PA 3110] sleeps 4 seconds.
[PB 3108] sleeps 2 seconds.
[PB 3109] sleeps 1 seconds.
[PA 3111] terminates.
[PB 3109] sleeps 1 seconds.
[PB 3108] sleeps 2 seconds.
[PB 3109] sleeps 1 seconds.
[PA 3110] terminates.
[PB 3109] sleeps 1 seconds.
[PA 3112] terminates.
[PB 3108] sleeps 2 seconds.

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

----- [MANAGER] Terminating running child processes -----
[MANAGER] Terminating PB process [3108]...
[MANAGER] Terminating PB process [3109]...
[PB 3108] terminated (SIGINT).
[PB 3109] terminated (SIGINT).
```

✂ Añote 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 2 3 4
```

Resultado:

Esqueleto de Código Fuente

A continuación se muestra el esqueleto de código fuente para resolver el ejercicio. Sólo debe incluir el código asociado al lanzamiento de un proceso y al tratamiento de argumentos de los procesos de tipo PA y PB.

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 pa pb
11
12 dirs:
13     mkdir -p $(DIROBJ) $(DIREXE)
14
15 manager: $(DIROBJ)manager.o
16     $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
17 pa: $(DIROBJ)pa.o
18     $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
19 pb: $(DIROBJ)pb.o
20     $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
21
22 $(DIROBJ)%.o: $(DIRSRC)%.c
23     $(CC) $(CFLAGS) $^ -o $@
24
25 test:
26     ./$(DIREXE)manager 3 2 5
27 solution:
28     ./$(DIREXE)manager 2 3 4
29
30 clean :
31     rm -rf *~ core $(DIROBJ) $(DIREXE) $(DIRHEA)*~ $(DIRSRC)*~
```

definitions.h

```
32 #define PA_CLASS "PA"
33 #define PA_PATH "./exec/pa"
34 #define PB_CLASS "PB"
35 #define PB_PATH "./exec/pb"
36
37 /* Process class */
38 enum ProcessClass_t {PA, PB};
39
40 /* Process info */
41 struct TProcess_t {
42     enum ProcessClass_t class; /* PA or PB */
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_by_class(enum ProcessClass_t class, int n new processes,
67                               int index_process_table, char *s_tmax_wait);
68 pid_t create_single_process(const char *path, const char *str_process_class,
69                             const char *arg);
69 void get_str_process_info(enum ProcessClass_t class, char **path, char **str_process_class);
70 void init_process_table(int nPA, int nPB);
71 void terminate_processes(void);
72 void wait_processes(int nPA);
```

```

73  /* Auxiliar functions */
74  void free_resources();
75  void install_signal_handler();
76  void parse_argv(int argc, char *argv[], int *nPA, int *nPB, char **s_tmax_wait);
77  void signal_handler(int signo);
78
79  /***** Main function *****/
80
81  int main(int argc, char *argv[]) {
82      char *s_tmax_wait = NULL;
83      int nPA, nPB;
84
85      parse_argv(argc, argv, &nPA, &nPB, &s_tmax_wait);
86      install_signal_handler();
87
88      init_process_table(nPA, nPB);
89      create_processes_by_class(PB, nPB, 0, s_tmax_wait);
90      create_processes_by_class(PA, nPA, nPB, s_tmax_wait);
91      wait_processes(nPA);
92
93      printf("\n[MANAGER] Program termination (all the PA processes terminated).\n");
94      terminate_processes();
95      free_resources();
96
97      return EXIT_SUCCESS;
98  }
99
100 /***** Process management *****/
101
102 void create_processes_by_class(enum ProcessClass t class, int n_new_processes,
103                               int index_process_table, char *s_tmax_wait) {
104     char *path = NULL, *str_process_class = NULL;
105     int i;
106     pid_t pid;
107
108     get_str_process_info(class, &path, &str_process_class);
109
110     for (i = index_process_table; i < (index_process_table + n_new_processes); i++) {
111         pid = create_single_process(path, str_process_class, s_tmax_wait);
112
113         g_process_table[i].class = class;
114         g_process_table[i].pid = pid;
115         g_process_table[i].str_process_class = str_process_class;
116     }
117
118     printf("[MANAGER] %d %s processes created.\n", number, str_process_class);
119     sleep(1);
120 }
121
122 pid_t create_single_process(const char *path, const char *str_process_class,
123                             const char *arg) {

```

✂ Incluya aquí el código de creación de un proceso (Aprox. ≈ 14 Líneas de código)

```

123 }
124
125 void get_str_process_info(enum ProcessClass t class, char **path,
126                           char **str_processes_class) {
127     switch (class) {
128     case PA:
129         *path = PA_PATH;
130         *str_processes_class = PA_CLASS;
131         break;

```

```

131     case PB:
132         *path = PB_PATH;
133         *str_process_class = PB_CLASS;
134         break;
135     }
136 }
137
138 void init_process_table(int nPA, int nPB) {
139     int i;
140
141     /* Number of processes to be created */
142     g_nProcesses = nPA + nPB;
143     /* Allocate memory for the 'process table' */
144     g_process_table = malloc(g_nProcesses * sizeof(struct TProcess_t));
145
146     /* Init the 'process table' */
147     for (i = 0; i < g_nProcesses; i++) {
148         g_process_table[i].pid = 0;
149     }
150 }
151
152 void terminate_processes(void) {
153     int i;
154
155     printf("\n----- [MANAGER] Terminating running child processes ----- \n");
156     for (i = 0; i < g_nProcesses; i++) {
157         /* Child process alive */
158         if (g_process_table[i].pid != 0) {
159             printf("[MANAGER] Terminating %s process [%d]...\n",
160                 g_process_table[i].str_process_class, g_process_table[i].pid);
161             if (kill(g_process_table[i].pid, SIGINT) == -1) {
162                 fprintf(stderr, "[MANAGER] Error using kill() on process %d: %s.\n",
163                     g_process_table[i].pid, strerror(errno));
164             }
165         }
166     }
167 }
168
169 void wait_processes(int nPA) {
170     int i;
171     pid_t pid;
172
173     /* Wait for the termination of PA processes */
174     while (nPA > 0) {
175         /* Wait for any PA process */
176         pid = wait(NULL);
177         for (i = 0; i < g_nProcesses; i++) {
178             if (pid == g_process_table[i].pid) {
179                 /* Update the 'process table' */
180                 g_process_table[i].pid = 0;
181                 /* Decrement the number of running PA processes */
182                 if (g_process_table[i].class == PA) {
183                     nPA--;
184                 }
185                 /* Child process found */
186                 break;
187             }
188         }
189     }
190 }
191
192 /***** Auxiliar functions *****/
193
194 void free_resources() {
195     /* Free the 'process table' memory */
196     free(g_process_table);
197 }
198
199 void install_signal_handler() {
200     if (signal(SIGINT, signal_handler) == SIG_ERR) {
201         fprintf(stderr, "[MANAGER] Error installing signal handler: %s.\n", strerror(errno));
202         exit(EXIT_FAILURE);
203     }
204 }
205
206 void parse_argv(int argc, char *argv[], int *nPA, int *nPB, char **s_tmax_wait) {
207     if (argc < 4) {
208         fprintf(stderr, "Error. Use: ./exec/manager <n_processes_PA> <n_processes_PB>
209             <t_max_wait>.\n");
210         exit(EXIT_FAILURE);
211     }
212
213     /* Number of PA/PB processes and max waiting time */
214     *nPA = atoi(argv[1]);
215     *nPB = atoi(argv[2]);
216     *s_tmax_wait = argv[3];
217 }
218
219 void signal_handler(int signo) {
220     printf("\n[MANAGER] Program termination (Ctrl + C).\n");
221     terminate_processes();
222     free_resources();
223     exit(EXIT_SUCCESS);
224 }


```

pa.c

```

224 #include <errno.h>
225 #include <signal.h>
226 #include <stdio.h>
227 #include <stdlib.h>
228 #include <string.h>
229 #include <unistd.h>
230
231 /* Program logic */
232 void run (int t_wait);
233
234 /* Auxiliar functions */
235 void install signal handler();
236 void parse_argv(int argc, char *argv[], int *t_wait);
237 void signal_handler(int signo);
238
239 /***** Main function *****/
240
241 int main (int argc, char *argv[]) {
242     int t_wait;
243
244     install signal handler();
245     parse_argv(argc, argv, &t_wait);
246
247     run(t_wait);
248
249     return EXIT_SUCCESS;
250 }
251
252 /***** Program logic *****/
253
254 void run(int t_wait) {
255     printf("[PA %d] sleeps %d seconds.\n", getpid(), t_wait);
256     sleep(t_wait);
257     printf("[PA %d] terminates.\n", getpid());
258 }
259
260 /***** Auxiliar functions *****/
261
262 void install signal handler() {
263     if (signal(SIGINT, signal_handler) == SIG_ERR) {
264         fprintf(stderr, "[PA %d] Error installing handler: %s.\n", getpid(), strerror(errno));
265         exit(EXIT_FAILURE);
266     }
267 }
268
269 void parse_argv(int argc, char *argv[], int *t_wait) {

```

 Incluya aquí el código de la función parse_argv (Longitud aprox. ≈ 6 Líneas de código)

```

270 }
271
272 void signal_handler(int signo) {
273     printf("[PA %d] terminated (SIGINT).\n", getpid());
274     exit(EXIT_SUCCESS);
275 }

```

pb.c

```

276 #include <errno.h>
277 #include <signal.h>
278 #include <stdio.h>
279 #include <stdlib.h>
280 #include <string.h>
281 #include <unistd.h>
282
283 /* Program logic */
284 void run(int t_wait);
285
286 /* Auxiliar functions */
287 void install_signal_handler();
288 void parse_argv(const int argc, char *argv[], int *t_wait);
289 void signal_handler(int signo);
290
291 /***** Main function *****/
292
293 int main(int argc, char *argv[]) {
294     int t_wait;
295
296     install_signal_handler();
297     parse_argv(argc, argv, &t_wait);
298
299     run(t_wait);
300
301     return EXIT_SUCCESS;
302 }
303
304 /***** Program logic *****/
305
306 void run(int t_wait) {
307     while(1) {
308         printf("[PB %d] sleeps %d seconds.\n", getpid(), t_wait);
309         sleep(t_wait);
310     }
311 }
312
313 /***** Auxiliar functions *****/
314
315 void install_signal_handler() {
316     if (signal(SIGINT, signal_handler) == SIG_ERR) {
317         fprintf(stderr, "[PB %d] Error installing handler: %s.\n", getpid(), strerror(errno));
318         exit(EXIT_FAILURE);
319     }
320 }
321
322 void parse_argv(int argc, char *argv[], int *t_wait) {

```

✂ Incluya aquí el código de la función `parse_argv` (Longitud aprox. ≈ 6 Líneas de código)

```

323 }
324
325 void signal_handler(int signo) {
326     printf("[PB %d] terminated (SIGINT).\n", getpid());
327     exit(EXIT_SUCCESS);
328 }

```