Faculty of Computer Science
University of Castilla-La Mancha

Concurrent and Real-Time Programming
Model A01 – Process Management

Lab activity

Model A01 – Process Management

LAST NAME: _____
NAME: _____ **LAB GROUP**: _____

Instructions:

- You cannot use books, notes nor mobile phones.
- When you have a working solution (compilation + execution), show it to the lecturer.
- You must write down the source code of your solution.

Grade

Statement

Build, using ANSI C, a system composed of **three executables** that simulates the following behavior. The system will be formed by three classes of processes: i) *manager*, ii) *PA* and iii) *PB*.

The ***manager* process** will be responsible for creating a number of processes of class PA and class PB, managing their termination and freeing the resources previously allocated.

On the one hand, the ***PA* processes** will simply *sleep* for a random number of seconds, between 1 and the value given by the first argument provided in the command-line. Next, they will terminate their execution.

On the other hand, the ***PB* processes** will run an infinite loop. In each iteration they will *sleep* for a random number of seconds, between 1 and the value given by the first argument provided in the command-line.

The number of processes of each class, along with the maximum waiting time, will be set by the user through the command line when executing the only *manager* process:

```
./exec/manager <n_processes_PA> <n_processes_PB> <t_max_wait>
```

where `<n_processes_PA>` represents the number of PA processes to be created, `<n_processes_PB>` the number of PB processes and, finally, `<t_max_wait>` represents the maximum waiting time of the processes (*sleeping* time), which will be given to the PA/PB processes in the creation time.

The **simulation will end** if one of the following situations occurs:

1. All the *PA* processes terminate. Upon detecting this situation, the *manager* process will terminate all the PB processes and will free the previously allocated resources.
2. The user types the 'Ctrl+C' key combination. Upon detecting this situation, the manager process will terminate all the PA/PB processes that are running and will free the previously allocated resources.

Resolution

Use the given source code to resolve the proposed exercise. This template code must not be modified. Include the required code in the indicated sections (frames).

Test example

Once a executable file has been generated, if you execute the following command (`make test`),

```
./exec/manager 3 2 5
```

the obtained result should be similar to the one shown below (the PIDs, the output sequence and the randomly generated values will be different):

```
[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).
```

🛠 Write down the last part of the obtained output when executing the following command (`make solution`), using the same output format that in the previous execution example:

```
./exec/manager 2 3 4
```

```
Result:
```

Source code template

Next, you can study the source code provided as a template for you to solve the exercise. **You must only include the code required to create a single process and the code required to handle the input arguments on behalf of PA/PB processes.**

Makefile

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

definitions.h

```
31   #define PA_CLASS "PA"
32   #define PA_PATH "./exec/pa"
33   #define PB_CLASS "PB"
34   #define PB_PATH "./exec/pb"
35
36   /* Process class */
37   enum ProcessClass_t {PA, PB};
38
39   /* Process info */
40   struct TProcess_t {
41     enum ProcessClass_t class; /* PA or PB */
42     pid_t pid;                 /* Process ID */
43     char *str_process_class;   /* String representation of the process class */
44   };
```

manager.c

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

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

🛠 Include the code required to create a process *(Aprox. ≈ 14 lines)*

```
122 }
123
124 void get_str_process_info(enum ProcessClass_t class, char **path,
                              char **str_process_class) {
125   switch (class) {
126   case PA:
127     *path = PA_PATH;
128     *str_process_class = PA_CLASS;
129     break;
```

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

pa.c

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

Include the code of the parse_argv function *(Aprox. ≈ 6 lines)*

```
269  }
270
271  void signal_handler(int signo) {
272    printf("[PA %d] terminated (SIGINT).\n", getpid());
273    exit(EXIT_SUCCESS);
274  }
```

pb.c

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

Include the code of the parse_argv function *(Aprox. ≈ 6 lines)*

```
322  }
323
324  void signal_handler(int signo) {
325    printf("[PB %d] terminated (SIGINT).\n", getpid());
326    exit(EXIT_SUCCESS);
327  }
```