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

```
DIROBJ := obj/
DIREXE := exec/
DIRHEA := include/
DIRSRC := src/
CFLAGS := -I$(DIRHEA) -c -Wall -ansi
LDLIBS := -lpthread -lrt
CC := gcc

all : dirs manager pa pb

dirs:
   mkdir -p $(DIROBJ) $(DIREXE)

manager: $(DIROBJ)manager.o
   $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
pa: $(DIROBJ)pa.o
   $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)
pb: $(DIROBJ)pb.o
   $(CC) -o $(DIREXE)$@ $^ $(LDLIBS)

$(DIROBJ)%.o: $(DIRSRC)%.c
   $(CC) $(CFLAGS) $^ -o $@

test:
   ./$(DIREXE)manager 3 2 5
solution:
   ./$(DIREXE)manager 2 3 4

clean :
   rm -rf *~ core $(DIROBJ) $(DIREXE) $(DIRHEA)*~ $(DIRSRC)*~
```

definitions.h

```
#define PA_CLASS "PA"
#define PA_PATH "./exec/pa"
#define PB_CLASS "PB"
#define PB_PATH "./exec/pb"

/* Process class */
enum ProcessClass_t {PA, PB};

/* Process info */
struct TProcess_t {
  enum ProcessClass_t class; /* PA or PB */
  pid_t pid;                 /* Process ID */
  char *str_process_class;   /* String representation of the process class */
};
```

manager.c

```
#define _POSIX_SOURCE

#include <errno.h>
#include <linux/limits.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>

#include <definitions.h>

/* Total number of processes */
int g_nProcesses;
/* 'Process table' (child processes) */
struct TProcess_t *g_process_table;

/* Process management */
void create_processes_by_class(enum ProcessClass_t class, int n_new_processes,
                               int index_process_table, char *s_tmax_wait);
pid_t create_single_process(const char *path, const char *str_process_class
                            const char *arg);
void get_str_process_info(enum ProcessClass_t class, char **path, char **str_process_class);
void init_process_table(int nPA, int nPB);
void terminate_processes(void);
void wait_processes(int nPA);
```

```c
/* Auxiliar functions */
void free_resources();
void install_signal_handler();
void parse_argv(int argc, char *argv[], int *nPA, int *nPB, char **s_tmax_wait);
void signal_handler(int signo);

/******************** Main function ********************/

int main(int argc, char *argv[]) {
  char *s_tmax_wait = NULL;
  int nPA, nPB;

  parse_argv(argc, argv, &nPA, &nPB, &s_tmax_wait);
  install_signal_handler();

  init_process_table(nPA, nPB);
  create_processes_by_class(PB, nPB, 0, s_tmax_wait);
  create_processes_by_class(PA, nPA, nPB, s_tmax_wait);
  wait_processes(nPA);

  printf("\n[MANAGER] Program termination (all the PA processes terminated).\n");
  terminate_processes();
  free_resources();

  return EXIT_SUCCESS;
}

/******************** Process management ********************/

void create_processes_by_class(enum ProcessClass_t class, int n_new_processes,
                               int index_process_table, char *s_tmax_wait) {
  char *path = NULL, *str_process_class = NULL;
  int i;
  pid_t pid;

  get_str_process_info(class, &path, &str_process_class);

  for (i = index_process_table; i < (index_process_table + n_new_processes); i++) {
    pid = create_single_process(path, str_process_class, s_tmax_wait);

    g_process_table[i].class = class;
    g_process_table[i].pid = pid;
    g_process_table[i].str_process_class = str_process_class;
  }

  printf("[MANAGER] %d %s processes created.\n", number, str_process_class);
  sleep(1);
}

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)*

```c
}

void get_str_process_info(enum ProcessClass_t class, char **path,
                          char **str_process_class) {
  switch (class) {
  case PA:
    *path = PA_PATH;
    *str_process_class = PA_CLASS;
    break;
  case PB:
    *path = PB_PATH;
    *str_process_class = PB_CLASS;
    break;
  }
}

void init_process_table(int nPA, int nPB) {
  int i;

  /* Number of processes to be created */
  g_nProcesses = nPA + nPB;
  /* Allocate memory for the 'process table' */
  g_process_table = malloc(g_nProcesses * sizeof(struct TProcess_t));

  /* Init the 'process table' */
  for (i = 0; i < g_nProcesses; i++) {
    g_process_table[i].pid = 0;
  }
}

void terminate_processes(void) {
  int i;

  printf("\n----- [MANAGER] Terminating running child processes ----- \n");
  for (i = 0; i < g_nProcesses; i++) {
    /* Child process alive */
    if (g_process_table[i].pid != 0) {
      printf("[MANAGER] Terminating %s process [%d]...\n",
             g_process_table[i].str_process_class, g_process_table[i].pid);
      if (kill(g_process_table[i].pid, SIGINT) == -1) {
        fprintf(stderr, "[MANAGER] Error using kill() on process %d: %s.\n",
                 g_process_table[i].pid, strerror(errno));
      }
    }
  }
}

void wait_processes(int nPA) {
  int i;
  pid_t pid;

  /* Wait for the termination of PA processes */
  while (nPA > 0) {
    /* Wait for any PA process */
    pid = wait(NULL);
    for (i = 0; i < g_nProcesses; i++) {
      if (pid == g_process_table[i].pid) {
        /* Update the 'process table' */
        g_process_table[i].pid = 0;
        /* Decrement the number of running PA processes */
        if (g_process_table[i].class == PA) {
          nPA--;
        }
        /* Child process found */
        break;
      }
    }
  }
}

/******************** Auxiliar functions ********************/

void free_resources() {
  /* Free the 'process table' memory */
  free(g_process_table);
}

void install_signal_handler() {
  if (signal(SIGINT, signal_handler) == SIG_ERR) {
    fprintf(stderr, "[MANAGER] Error installing signal handler: %s.\n", strerror(errno));
    exit(EXIT_FAILURE);
  }
}

void parse_argv(int argc, char *argv[], int *nPA, int *nPB, char **s_tmax_wait) {
  if (argc < 4) {
    fprintf(stderr, "Error. Use: ./exec/manager <n_processes_PA> <n_processes_PB>
                     <t_max_wait>.\n");
    exit(EXIT_FAILURE);
  }

  /* Number of PA/PB processes and max waiting time */
  *nPA = atoi(argv[1]);
  *nPB = atoi(argv[2]);
  *s_tmax_wait = argv[3];
}

void signal_handler(int signo) {
  printf("\n[MANAGER] Program termination (Ctrl + C).\n");
  terminate_processes();
  free_resources();
  exit(EXIT_SUCCESS);
}
```

pa.c

```c
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

/* Program logic */
void run (int t_wait);

/* Auxiliar functions */
void install_signal_handler();
void parse_argv(int argc, char *argv[], int *t_wait);
void signal_handler(int signo);

/******************** Main function ********************/

int main (int argc, char *argv[]) {
  int t_wait;

  install_signal_handler();
  parse_argv(argc, argv, &t_wait);

  run(t_wait);

  return EXIT_SUCCESS;
}

/******************** Program logic ********************/

void run(int t_wait) {
  printf("[PA %d] sleeps %d seconds.\n", getpid(), t_wait);
  sleep(t_wait);
  printf("[PA %d] terminates.\n", getpid());
}

/******************** Auxiliar functions ********************/

void install_signal_handler() {
  if (signal(SIGINT, signal_handler) == SIG_ERR) {
    fprintf(stderr, "[PA %d] Error installing handler: %s.\n", getpid(), strerror(errno));
    exit(EXIT_FAILURE);
  }
}

void parse_argv(int argc, char *argv[], int *t_wait) {
```

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

```c
}

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

pb.c

```
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

/* Program logic */
void run(int t_wait);

/* Auxiliar functions */
void install_signal_handler();
void parse_argv(const int argc, char *argv[], int *t_wait);
void signal_handler(int signo);

/******************** Main function ********************/

int main(int argc, char *argv[]) {
  int t_wait;

  install_signal_handler();
  parse_argv(argc, argv, &t_wait);

  run(t_wait);

  return EXIT_SUCCESS;
}

/******************** Program logic ********************/

void run(int t_wait) {
  while(1) {
    printf("[PB %d] sleeps %d seconds.\n", getpid(), t_wait);
    sleep(t_wait);
  }
}

/******************** Auxiliar functions ********************/

void install_signal_handler() {
if (signal(SIGINT, signal_handler) == SIG_ERR) {
    fprintf(stderr, "[PB %d] Error installing handler: %s.\n", getpid(), strerror(errno));
    exit(EXIT_FAILURE);
  }
}

void parse_argv(int argc, char *argv[], int *t_wait) {
```

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

```
}

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