F-ITR303 – Real Time Systems (IN)

Final Project - Minepump

By

Xiao Niu

Present to

M. Youcef Bouchebaba

Mme. Ahlem Mifdaoui

13/12/2019

Exercise 1: *minteger* implementation

In this first exercise, there are two things to be done: the first consists of the initialization of the ***m_integer*** structure, and the second consists of the implementation of the read/write operations defined in the file.

Let's start with the initialization of the structure, which is defined inside the function ***MI_init()***. Here the structure has been created and been named to ***m*** with the type ***m_integer*** defined in the file [*.h*]. There are two attributes for this structure including an integer type ***value*** and a mutex. The attribute ***value*** has already been initialized with 0, so we can simply initialize the mutex with the primitive ***pthread_mutex_init()*** by using the default mutex created inside the structure type ***m_integer***. The ***CHECK_NZ()*** function is used in order to make sure that there is no error when the function is being executed.

Then we have the two functions to be completed: ***MI_write()*** and ***MI_read()***. In ***MI_write()***, the input argument ***v*** is assigned to the structure ***m***, and in ***MI_read()*** the value of ***v*** can be obtained; here we want to ensure that both operations must not be interrupted during their execution, so we use the methods ***pthread_mutex_lock()*** and ***pthread_mutex_unlock()*** to protect these parts of the code. As in the previous job, the function ***CHECK_NZ()*** is used again for verification.

At the end, with the test defined in the ***main*** function, the execution of ***test_minteger*** gives the expected output:

```
[x.niu@pcb61-01-02 minepump]$ ./test_minteger
Read 42
```

which allows to validate this first exercise.

Exercise 2: *msg_box* implementation

In this second exercise, there are also two things to be done: in a similar way comparing to the previous question, the first job is to initialize the ***msg_box*** structure, and the second is to implement the send/receive operations defined in the file.

Starting with the initialization part: this is defined inside de function ***msg_box_init()***. Here the structure has been created and been named to ***mbox*** with the type ***msg_box*** defined in the file [*.h*]. The attributes related to the size and to the buffer have already been initialized, so the job consists of creating the mutex and the condition variable by using respectively the two following methods: ***pthread_mutex_init()*** and ***pthread_cond_init()***. As in the previous question, the function ***CHECK_NZ()*** is used for verification.

Now we need to complete the functions: ***msg_box_receive()*** and ***msg_box_send()***. The concurrency control is always done by using mutex. In addition, now with the introduction of a condition variable, on one hand we define the moment once the send function finishes its critical section by sending a signal (with the method: ***pthread_cond_signal()***) to the receive function. And on the other hand, we

suspend this receive function (with the method ***pthread_cond_wait()***) until being waked up by the signal. In this way, the atomicity is guaranteed.

At the end, with the test defined in the ***main*** function, the execution of ***test_msgbox*** gives the output expected:

```
[x.niu@pcb61-01-02 minepump]$ ./test_msgbox
msg_box_send: 1
Read a
```

,

which allows to validate this second exercise.

Exercise 3: Implementing periodic tasks

In this third exercise, there are two functions to be implemented: ***periodic_task_body()*** and ***create_periodic_task()***.

The function ***periodic_task_body()*** is used to initialize the structure of periodic tasks. Once a task is created, a semaphore value of 0 is assigned to it (with the function ***sem_init(&timer,0,0)***), so the task is initially blocked. Then, we use the function ***clock_gettime(CLOCK_REALTIME, &trigger)*** and an infinite loop to wake up the task after receiving the trigger.

```
sem_init(&timer,0,0);
clock_gettime(CLOCK_REALTIME, &trigger);
for(;;){
  (*my_parameters).job();
  add_timespec (&trigger, &trigger, &period);
  sem_timedwait(&timer, &trigger);
}
```

The function ***create_periodic_task()*** uses a thread and the previous function to create the task. At first, the attributes ***period*** and ***job*** of the ***parameter*** pointer are initialized, and then the attribute of the thread ***attr*** is defined (with ***pthread_attr_t*** and ***pthread_attr_init()***). Then, the periodic task is created by using the method ***pthread_create()*** :

```
pthread_create (&tid,&attr,periodic_task_body,  (void *)parameters);
```

At the end, with the test defined in the ***main*** function, the execution of ***test_periodic*** gives the output expected:

```
[x.niu@pcb61-01-02 minepump]$ ./test_periodic
Task created
o<
o<
o<
o<
o<
o<
o<
```

,

which allows to validate this third exercise.

Exercise 4: Running the minepump

In this fourth exercise, our job is to implement the four tasks of the pump (***WaterLevelMonitoring***, ***MethaneMonitoring***, ***PumpCtrl***, and ***CmdAlarm***) and to complete the setup for the simulation.

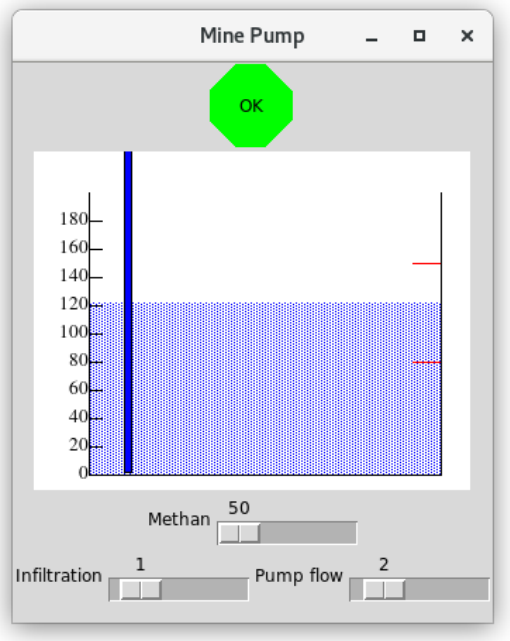
Firstly, the body of each task is completed with the instructions inside the comments:

- ***WaterLevelMonitoring_Body***:
  - This periodic task reads the water level of the mine with the functions (***ReadHLS()*** and ***ReadLLS()***), then it updates the value of the ***m_integer*** type variable : ***LvlWater*** with the function ***MI_write()***.
- ***MethaneMonitoring_Body***:
  - This periodic task reads the methane level of the mine with the function ReadMS(), then it updates the value of the ***m_integer*** type variable : ***LvlMeth*** with the function ***MI_write()***. This tasks also triggers the semaphore ***synchro*** by using the method ***sem_post()***.
- ***PumpCtrl_Body***:
  - This sporadic task is triggered by the semaphore posted by the task ***MethaneMonitoring***, then it reads the values of the variables ***LvlWater*** *and* ***LvlMeth***. Depending on the levels of the water and methane detected, this function updates the value of the variable ***cmd*** and controls the operation of the pump with ***CmdPump(cmd***).
- ***CmdAlarm_Body***:
  - This sporadic task has already been completed; no further actions are required.

Secondly, we finish the setup for the simulation in the ***main()*** function. The first things to do is the initialization of the primitives (semaphore, control variables, etc.). Then, the four tasks are created in order. For each periodic task (***WaterLevelMonitoring*** and ***MethaneMonitoring***), its period is defined within the ***timespec*** structure and the task is directly created by using the function ***create_periodic_task(period, body)***, where the body argument calls the functions defined previously. For each sporadic task (***PumpCtrl*** and ***CmdAlarm***), we use a thread to create the task by calling its corresponding body function inside the method ***pthread_create()***. This marks the end of the design of our pump.

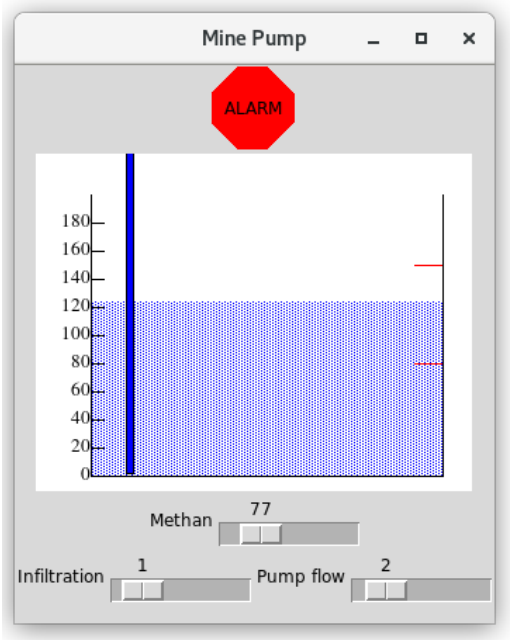
At the end, we use the ***TCL/TK GUI*** to simulate graphically the mine and to run in parallel our pump after compilation. The following scenarios have been observed:

Scenario 1 :



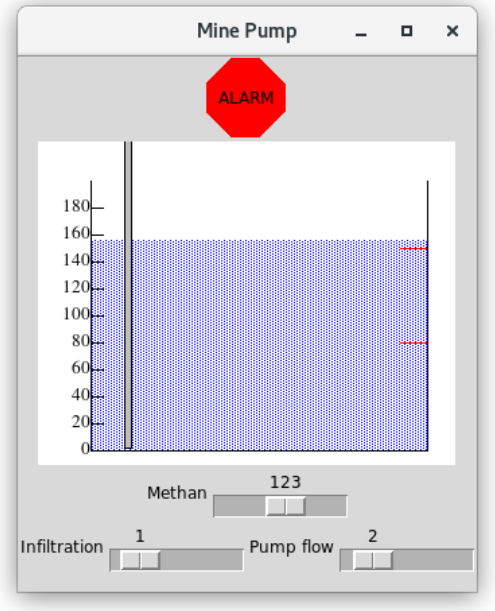
In this scenario, the water firstly reaches the High-level, then the pump is activated to bring back the water to the Low-level. The Methane level is alright, so the alarm is not triggered.

Scenario 2:



In this Scenario, the methane level increases and becomes greater than the first threshold. The Alarm is triggered, but the pump is still working (as the methane level has not reached the second threshold yet).

Scenario 3:



In this scenario, the methane level becomes greater than its second threshold. Therefore, the pump is no longer working even though the water level is above the High-level line.

As these results correspond to the expectations, the design of our mine pump is validated.

Exercise 5: Removing the GUI, moving to RTEMS

In this last exercise, the program is simulated in two different ways: firstly by removing the ***TCL/TK GUI***, secondly by being moved to ***RTEMS***.

With the ***Makefile*** provided, the execution file ***minepump _nogui*** is generated. With the scenario described in the file ***simu.c***, the following simulation is obtained (only a few lines are shown):

```
ML = 066        WL = 056        Pump 0  Alarm 0
ML = 068        WL = 058        Pump 0  Alarm 0
ML = 070        WL = 058        Pump 0  Alarm 0
ML = 072        WL = 060        Pump 0  Alarm 1
ML = 074        WL = 060        Pump 0  Alarm 1
ML = 076        WL = 060        Pump 0  Alarm 1
ML = 078        WL = 062        Pump 0  Alarm 1
ML = 080        WL = 062        Pump 0  Alarm 1
```

```
ML = 064        WL = 104        Pump 1  Alarm 0
ML = 066        WL = 104        Pump 1  Alarm 0
ML = 068        WL = 102        Pump 1  Alarm 0
ML = 070        WL = 102        Pump 1  Alarm 0
ML = 072        WL = 100        Pump 1  Alarm 1
ML = 074        WL = 100        Pump 1  Alarm 1
ML = 076        WL = 100        Pump 1  Alarm 1
```

Then, by moving the program to the ***Prise*** server, we run the scenario with ***RTEMS***:

```
[x.niu@prise-srv2 minepump]$ source ../rtems_lab/rtems.env
[x.niu@prise-srv2 minepump]$ make minepump_rtems
```

```
[x.niu@prise-srv2 minepump]$ make rtems_run
tsim-leon3 o-optimize/minepump.exe
```

and same results have been obtained (only a few lines are shown):

```
ML = 066        WL = 056        Pump 0  Alarm 0
ML = 068        WL = 058        Pump 0  Alarm 0
ML = 070        WL = 058        Pump 0  Alarm 0
ML = 072        WL = 060        Pump 0  Alarm 1
ML = 074        WL = 060        Pump 0  Alarm 1
ML = 076        WL = 060        Pump 0  Alarm 1
ML = 078        WL = 062        Pump 0  Alarm 1
ML = 080        WL = 062        Pump 0  Alarm 1

ML = 064        WL = 152        Pump 1  Alarm 0
ML = 066        WL = 152        Pump 1  Alarm 0
ML = 068        WL = 150        Pump 1  Alarm 0
ML = 070        WL = 150        Pump 1  Alarm 0
ML = 072        WL = 148        Pump 1  Alarm 1
ML = 074        WL = 148        Pump 1  Alarm 1
ML = 076        WL = 148        Pump 1  Alarm 1
```

By analyzing these results, we see that the pump works as expected:

- When both levels (methane and water) are below their first threshold, neither the pump nor the alarm is activated;
- When the methane is above its first threshold and the water level is low, only the alarm activates;
- When the methane level is low and the water is above its second threshold, only the pump is activated;
- When the methane level is above its second threshold and the water level is above its second threshold, both the pump and the alarm are activated;

Consequently, this allows to validate this last exercise of this project.