Rapport ENSTA 2021 - ROB305 - Kai Zhang & Yan CHEN

This report contains the answers to all td's and explanations on course ROB305. It covers two aspects : 1.tmeps management 2. multi-task programming. POSIX API is used in this project.

[TD-1] Mesure de temps et échantillonnage en temps

Library ***<time.h>*** and ***<signal.h>*** are used in this TD. Some functions of time measurement and of time sampling are implemented.

a) Simplified time management Posix

This part is intended to produce functions and operators for simple use of the ***timespec*** structure representing time measurement in the POSIX API. The timespec structure is made up of two parts. The one is ***tv_sec*** presenting seconds. The other is **tv_nsec** presenting nanoseconds.

```
double timespec_to_ms(const timespec& time_ts);
timespec timespec_from_ms(double time_ms)
timespec timespec_now();
timespec timespec_negate(const timespec& time_ts);
timespec timespec_add(const timespec& time1_ts, const timespec&
       time2_ts);
timespec timespec_subtract(const timespec& time1_ts, const
       timespec& time2_ts);
timespec timespec_wait(const timespec& delay_ms);
```

The above functions are implemented.

- ***timespec_to_ms*** can convert timespec type of time to time in milisecond.
- ***timespec_from_ms*** has the inverse functions ***with timpsec_to_ms***.
- ***timespec_now*** obtain current time
- ***timespec_negate*** obtain the opposite of a time
- ***timespec_add*** can add two times together
- ***timespec_subtract*** allows two times to be subtracted from each other
- ***timsepec_wait*** allows program wait delay_ms milliseconds

Some operator overloading is implemented in this part, like operation **-** and operation **+**. It can respectively allow two times to be subtracted and to be added.

Test result :

```
delay in ms: 500
delay in timespec:0s,500000000ns
successful
start time 1617520073,675574231
end time 1617520074,175802522
negative end time-1617520073s,824197478ns
- end time-1617520073s,824197478ns
duration 0,500228291
end > start
end != start
start+duration=end
```

b) Timers with callback

In this part, a POSIX timer with 2HZ was created that can print periodically a message with the value of a regularly incremented counter. Here we set it to 15 counters.

Two functions ***handler*** and ***generate_timer*** were implemented. In the function generate_timer, we declared the ***sigaction*** type of variable to set the attribution of signal, including call back function ***handler***, mask signal, etc. The ***sigevent*** type of variable can set the signal event, including signal number (SIGRTMIN), output value of signal, etc. The ***itimerspec*** type of variable ***its*** is used to set the period of signal. ***its.it_value*** set expiry time of timer. Here we set 0.5 seconds. ***its.it_interval*** reset the period of signal when the timer expiry. Here we set the same value with ***its.it_value*** 0.5 seconds. Function ***handler*** is the callback function. When one period expires, the counter increases one.

Here is the result of the test. It needs 7.5 seconds to increase 15 counters.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
----Duration----
7.50003 seconds
```

c) Simple CPU consuming function

In this part, we define signature function ***void incr***. This function performs a loop incrementing the counter value pointed to by ***pCounter*** by 1.0. It performs this loop ***nLoops*** times.

```
void incr(unsigned int nLoops, double* pCounter)
{
   for(int i=0; i<nLoops; i++)
   {
      *pCounter += 1,0;
   }
}
```

In the code of test, we use ***int main(int argc, char* argv[])*** with two parameters ***argc*** and ***argv***, which can receive the parameters that users enter when they run the executable fichier. In the test code, we use ***stroi(argv[1])*** function to receive the parameters that users enter.

The function ***timespec_now()*** defined in the TD1 a) has been used to record the beginning time of program and the end time of program, so that we can calculate the execution time of the program with end time - beginning time..

Here is the result of the test on the raspberry pi. We set ***nLoops*** 1000000, the execution time is 0.037867 seconds.

```
rob305-pi43# ./td1c 1000000
final value:1e+06
running time is 0.0367867
```

d) Measuring the execution time of a function

In this part, we combine ***generate_timer*** function and ***void incr*** function to redefine function ***unsigned incr(unsigned int nLoops, double* pCounter, bool* pStop)*** and function ***void incr_handler(int , siginfo_t* si, void*)*** to calculate the number of loops during a fixed time. callback function is incr_handler and the sigevent variable ***sev.sigev_value.sival_ptr*** points to the address of the bool ***stop*** variable. Bool ***stop*** become true in a setted fixed time. Then it stops the counter.

```
void incr_handler(int , siginfo_t* si, void* )
{
   bool* pStop=(bool*)(si->si_value.sival_ptr);
   *pStop=true;
}
```

```
unsigned incr(unsigned int nLoops, double* pCounter, bool* pStop)
{
   unsigned realLoops = 0;
   for(unsigned int iLoop=0; iLoop < nLoops; ++iLoop)
   {
      if(*pStop) break;
      *pCounter += 1.0;
      realLoops += 1;
   }
   return realLoops;
}
```

We assume that the number of loops and the execution time are linearly related $l(t) = at + b$. $l(t)$is the number of loops performed by the incr function during the time interval t. We define a function ***calib*** to calculate the value a and b.

```
void calib(double* a, double* b, double n1, double n2, double t1, double t2)
{
   *a=(n1-n2)/(t1-t2+1e-9); // in case that t1=t2
   *b=(n1-*a*t1);
}
```

Here is the result of the test on the raspberry pi. We can see that the error is very small after calibration.

```
a=4.16668e+08,b=6.8874e+07
the expected number of loops is 2152218771
the actual number of loops is 2146606800
the difference is 5611971
the error is 0.261435%
```

e) Improvement of the measures

We created a function ***calib*** which implemented the linear regression method to calculate the coefficients ***a*** and ***b*** based on multiple measurements. The accuracy is improved by this new calibration method.

Here is the result of the test on the raspberry pi. We can see that the error is quite small and less than the previous error.

```
rob305-pi43# ./td1e
test 1s
test 2s
test 3s
test 4s
test 5s
Mesurement times : 5
a=1.85704e+07,b=-21474
the expected number of loops is 92832111
the actual number of loops is 92846783
the difference is 14672
the error is +-0.0158024%
```

[TD-2] Familiarisation avec l'API multitâches *pthread*

a) Exécution sur plusieurs tâches sans mutex

In this part, several threads were created to execute their tasks respectively. We used the same task like td1 b). Id of threads were defined the type of ***vector<pthread>***. These threads were created one by one. After they finished their task, they were destroyed one by one. At the same time, we defined the function ***void* call_incr*** with input parameter structure ***Data{unsigned loop; double counter;}*** to call function ***void incr*** defined previously.

```
vector<pthread_t> incrThread(nTasks);
for(int i=0;i<nTasks;i++)
{
    pthread_create(&incrThread[i], nullptr, call_incr, &data);
}

for(int i=0;i<nTasks;i++)
{
    pthread_join(incrThread[i], nullptr);
}
```

```
void* call_incr(void* v_data)
{
    Data* pData=(Data*) v_data;
    incr(pData->loop, &(pData->counter));
    return v_data;
}
```

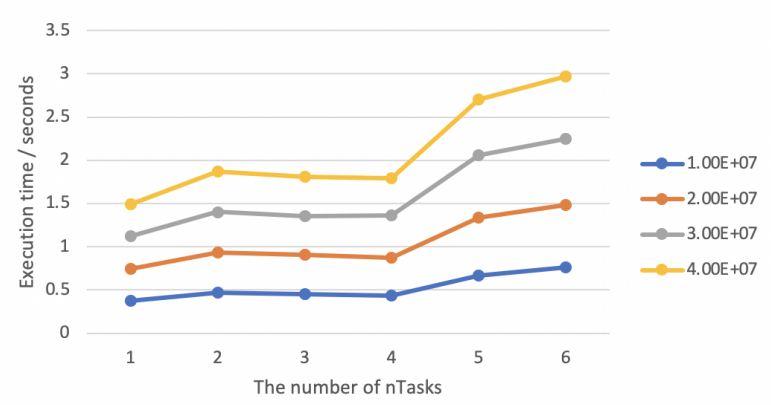
Here is the result of the test on raspberry pi. The first parameter is the number of loops and the second parameter is the number of threads. We can see that the total number of loops is the sum of the number of loops of each thread times the number of threads.

```
rob305-pi43# ./td2a 1000 5
counter value: 5000
rob305-pi43# ./td2a 1000 6
counter value: 6000
rob305-pi43# ./td2a 1000 7
counter value: 7000
```

b) Mesure de temps d'exécution

In this part, we measured the execution time of different tasks and loops. SCHED_RR thread policy was used.The result of the test as follows.

nTasks \ nLoops	1.00E+07	2.00E+07	3.00E+07	4.00E+07
1	0.37452	0.748255	1.12239	1.49574
2	0.468059	0.93439	1.4019	1.86878
3	0.453036	0.905219	1.3583	1.8104
4	0.438871	0.877284	1.36317	1.796
5	0.671079	1.34052	2.05605	2.70375
6	0.764461	1.48646	2.24893	2.97231



From this table we can see that, when the total number of loops, which is equal to **nLoops×nTasks,** multiple tasks runs more efficiently. However, it will cause the wrong count number, which is different to the expected value (**nLoops×nTasks).**

c) Exécution sur plusieurs tâches avec mutex

When we add a mutex to the counter, the processing time increases greatly with the same number of loops and tasks. Here is an example.

- When nLoops=1E+07, nTasks=2, if there is no mutex, the count value is 1.08177e+07 and the time is 43.6357ms.
- When there is a mutex, the count value is 2e+07 and the time is 677.617ms.

In conclusion, the mutex could permit the successful read and write of a value for multiple threads, but it increases the running time of the loops.

[TD-3] Classes pour la gestion du temps

a) Classe Chrono

In this part, we define a class Chrono by the function defined in TimeUtils.h. It can calculate the execution time of a program. The function ***lap()*** can calculate the duration of the program from the objective of class Chrono generated to ***stop()*** function called or the ***lap()*** called.

Here is the result of the test.

```
wait time 1s ...
done
duration 1.00008s
```

b) Classe Timer

Normally, constructor and destructor are public. They can't be derived by child class. For class timer, function ***start()*** and ***stop()*** should be public. They will be called outside of the class. Specifically, ***start()*** function should be virtual because it will be overridden in child class PeriodicTimer. ***callback()*** is protected and pure virtual because it must be implemented by child class (CountDown). For the class PeriodicTimer, the ***start()*** is overridden and will be called by the instance outsides, so it is virtual and public.

call_callback() function calls ***callback()*** function that realises the counter. It is private and static so it can only be called inside of the class. Besides, it doesn't need to create an instance to call this function.

The result of the test is as follows. The counter decreases by one per second.

```
timer: 9
timer: 8
timer: 7
timer: 6
timer: 5
timer: 4
timer: 3
timer: 2
timer: 1
timer: 0
--------stop------
```

c) Calibration en temps d'une boucle

We defined a Calibrator class to calculate the coefficients in function l(t)=at+b. It applies a linear regression algorithm to calculate a and b based on multiple measurements. After obtaining a and b, we set a test time t=1000ms. We get the estimated loop numbers through the l(t) function. On the other hand, we run the cpuLoop for 1000ms and obtain the real loops. Finally, we compare the difference between the estimated loop number and real loop number.

The result of the test is as follows.

```
----------stop-----------
a= 17106.5,b= -8345.88
finished calibration
running time 999.581
The real number of loop for1000ms is 17098143
The estimated number of loop for1000ms is 17098142
The difference is 1
```

From the result, we can see that the difference is one loop, which is very small and can be nearly ignored.

[TD-4] Classes de base pour la programmation multitâche

a) Classe Thread

We defined a class named IncrThread, which is derived from class Thread. It doesn't contain the mutex and each instance was assigned with a random priority. The schedule policy is SCHED_RR. We test this program by applying 10000000 loops using 4 threads. The final result is shown as follows.

```
rob305-pi59# ./td4a.rpi2 10000000 4
thread 0 time is 591.563
thread 1 time is 581.715
thread 2 time is 579.461
thread 3 time is 544.334
counter value: 1.61465e+07
```

From the result, we can see that, without a mutex, each thread runs independently and modifies the counter value randomly, which causes the unexpected counting value.

b) Classes Mutex et Mutex::Lock

We defined a class named IncrMutex, which contains un mutex to lock the counter when a thread tries to change its value. Then, we test our program by using 10 threads and each of them run 10000000 loops. The result is shown as follows.

```
rob305-pi59# ./td4b.rpi2 10000000 10
thread 0 time is 492.936
thread 1 time is 3424.44
thread 2 time is 1469.83
thread 3 time is 976.773
thread 4 time is 4401.45
thread 5 time is 1953.59
thread 6 time is 2930.74
thread 7 time is 3907.74
thread 8 time is 2446.21
thread 9 time is 4884.85
counter value: 1e+08
```

From the final counter value, we can see that it is equal to the number of loops times the number of tasks, which confirms that our programs runs correctly.

c) Classe Semaphore

We defined three classes here, including Semaphore class, Consumer class and Producer class. The Semaphore object has an initial number of chips(jeton) and it can contain at most 1000 items. The Consumer class will take an item every second and the Producer can produce at most 10000 items. When the semaphore is empty, the consumer can not take any item and it will wait until the producer adds new items. When the number of contained items reaches the maximum capability, the producer will stop adding and wait for the consumption.

We test the program by using 10 consumers and 10 producers. The result is shown as follows.

```
total consumption is 10000
total production is 10000
the difference is 0
```

From the result we can see that the total number of consumption is equal to the production, which means that this program well considers the capability of the semaphore.

d) Classe Fifo multitâches

We created three classes to solve this problem, one is ***Fifo*** and the other two are ***Consumer*** and ***Producer***. Consumer class aims to pop up the first element of the queue and the Producer class adds a new element to the queue every second. Each producer can add at most 1000 elements to the queue. When the queue is empty, the Consumer will stop working and wait until the producers add new elements.

We test the program by using 10 consumers and 10 producers. The result is shown as follows.

```
total consumption is 10000
total production is 10000
the difference is 0
```

From the result, we can see that the total number of consumption is the same as the number of production, which confirms that our program achieves our goal successfully.

[TD-5] Inversion de priorité

- **Soient A, B et C trois tâches, et R une ressource telles que**
  - priorité(A) > priorité(B) > priorité(C)
  - R est accédée par les tâches A et C
  - **t** est le temps en tics système, **t** ∈ {0, 1, 2, …}
- **Caractéristiques de la tâche A**
  - la tâche A est activée à **t** = 30
  - le temps d'exécution de A est de 40 tics
  - le délai de A est de 60 tics
  - la tâche A demande l'accès à R après 10 tics d'exécution
  - après avoir obtenu l'accès à R, A libère l'accès à R au bout de 10 tics d'exécution
- **Caractéristiques de la tâche B**
  - la tâche B est activée à **t** = 30
  - le temps d'exécution de B est de 10 tics
  - le délai de B est de 70 tics
- **Caractéristiques de la tâche C**
  - la tâche C est activée à **t** = 0
  - le temps d'exécution de C est de 50 tics
  - le délai de C est de 110 tics
  - la tâche C demande l'accès à R après 20 tics d'exécution
  - après avoir obtenu l'accès à R, C libère l'accès à R au bout de 20 tics d'exécution

We implemented the scheduling of three threads, whose configurations are shown in the above figure. The scheduling result is shown as follows.

```
rob305-pi59# ./td5.rpi2
----------stop-----------
a= 16793.1,b= -2810.91
successful
priority 11 starts thread
priority 11 asks for mutex
successful
priority 33 starts thread
priority 33 asks for mutex
successful
priority 22 starts thread
priority 22 terminates
priority 33 releases mutex
priority 33 terminates
priority 11 releases mutex
priority 11 terminates
total running time 9989.98ms
thread A took 8989.74ms
thread B took 4997.41ms
thread C took 9988.42ms
```