Pierre Kurkdjian September 2018

GMTO, Pasadena

**Developing Preempt-RT applications**

**Installation**

Preempt-RT patch is already installed on gposimrtc03 (172.16.10.13 via ssh).

If Preempt-RT is not installed on your Linux computer, you will find the installation instructions at this webpage:

wiki.linuxfoundation.org/realtime/documentation/howto/applications/preemptrt\_setup

Make sure the path is properly installed on your machine by entering the following command:

[pkurkdjian@gposimrtc03 ~]$ uname -v

Preempt-RT must appear in the command response:

#1 SMP PREEMPT RT Thu Jul 5 21:24:51 UTC 2018

**Important functions**

The following function is defined in <[sys/mman.h](https://linux.die.net/include/sys/mman.h)>

|  |  |
| --- | --- |
| Function | int **mlockall**(int flags) |
| Description | mlockall() prevents the OS from saving the current process in a swap area (in the hard disk). The process can be store in RAM only (to improve time performance). |
| Arguments | int flags: MCL\_CURRENT and MCL\_FUTURE determine whether the pages to be locked are those currently mapped, or the pages mapped in the future. |
| Output | 0 on success, -1 on error |

The next 5 functions set the attribute of the thread. The attribute must be previously declared and will be used to create the new thread. These functions are defined in <[pthread.h](http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html)> and the for the thread attribute identifiers in <sched.h>

|  |  |
| --- | --- |
| Function | int **pthread\_attr\_init**(pthread\_attr\_t \*attr) |
| Description | pthread\_attr\_init() initialises a thread attributes object attr with the default value. |
| Arguments | pthread\_attr\_t \*attr: a pointer to a thread attribute. |
| Output | 0 on success, returns a value different from 0 on error |

|  |  |
| --- | --- |
| Function | int **pthread\_attr\_setstacksize**(pthread\_attr\_t \*attr, size\_t stacksize) |
| Description | pthread\_attr\_setstacksize() set the maximum size of the thread. |
| Arguments | pthread\_attr\_t \*attr: a pointer to a thread attribute.  size\_t stacksize: the maximum size of the thread (in bytes) |
| Output | 0 on success, returns a value different from 0 on error |

|  |  |
| --- | --- |
| Function | int **pthread\_attr\_setschedpolicy**(pthread\_attr\_t \*attr, int policy) |
| Description | pthread\_attr\_setschedpolicy () set the maximum size of the thread. |
| Arguments | pthread\_attr\_t \*attr: a pointer to a thread attribute.  int policy:   * SCHED\_FIFO * SCHED\_RR (round robin) * SCHED\_OTHER   SCHED\_FIFO or SCHED\_RR are waiting on a mutex, they acquire the mutex in priority order when the mutex is unlocked |
| Output | 0 on success, returns a value different from 0 on error |

|  |  |
| --- | --- |
| Function | int **pthread\_attr\_setschedparam**(pthread\_attr\_t \*attr,  const struct sched\_param \*param) |
| Description | pthread\_attr\_setschedparam () set the scheduling parameter of the thread. |
| Arguments | pthread\_attr\_t \*attr: a pointer to a thread attribute.  sched\_param \*param: a pointer on a scheduling parameter  For the SCHED\_FIFO and SCHED\_RR policies, the only required member of param is sched\_priority.  param.sched\_priority between 1 (lowest priority) and 99 (highest priority) |
| Output | 0 on success, returns a value different from 0 on error |

|  |  |
| --- | --- |
| Function | int **pthread\_attr\_setinheritsched**(pthread\_attr\_t \*attr,  int inheritsched) |
| Description | pthread\_attr\_setinheritsched () set the inheritsched attribute  of the thread. This determines whether the other scheduling attributes of the created thread are inherited or not. |
| Arguments | pthread\_attr\_t \*attr: a pointer to a thread attribute.  int inheritsched:   * PTHREAD\_INHERIT\_SCHED: the thread attributes are to be inherited from the creating thread * PTHREAD\_EXPLICIT\_SCHED: the threadattributes are to be set to the corresponding values from this attribute object |
| Output | 0 on success, returns a value different from 0 on error |

The next 2 functions control the thread. They are defined in <[pthread.h](http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html)>

|  |  |
| --- | --- |
| Function | int **pthread\_create**(pthread\_t \*thread, const pthread\_attr\_t \*attr,  void \*(\*start\_routine)(void\*), void \*arg) |
| Description | pthread\_create() creates a new thread. This thread exists until the start\_routine function returns or pthread\_exit() is called. |
| Arguments | pthread\_t \*thread: the thread ID will be stored here  const pthread\_attr\_t \*attr: the attribute of the thread  void \*(\*start\_routine)(void\*): a pointer on the function called by the thread  void \*arg: a pointer on the function arguments |
| Output | 0 on success, returns a value different from 0 on error |

|  |  |
| --- | --- |
| Function | int **pthread\_join**(pthread\_t thread, void \*\*retval) |
| Description | pthread\_join () waits until the created thread terminates. If it has already terminated, pthread\_join () returns immediately. |
| Arguments | pthread\_t thread: the created thread  void \*\*retval: if not null, the exit status of the created thread is copied here. |
| Output | 0 on success, returns a value different from 0 on error |

**Implementation**

The source code files are committed on my GitHub account: github.com/pierrekurkdjian/preempt-RT

A real-time program with Preempt-RT is basically developed according

1. **Declaration**

First of all the necessary library must be included. The following libraries are necessary:

* pthread.h (thread management functions)
* sched.h and limits.h (thread management constants)
* sys/mman.h (memory management)

#include <limits.h>

#include <pthread.h>

#include <sched.h>

#include <sys/mman.h>

Then the variables related to the thread must be declared

pthread\_t thread;

struct sched\_param param;

pthread\_attr\_t attr;

1. **Attributes and schedule parameters settings**

Before its creation, the new thread must be parameterized.

The pthread attributes define the main main features of the new thread such as the stack size and the stack address of the thread when it’s stored in the RAM for example. The list of the parameters is given bellow:

|  |  |  |
| --- | --- | --- |
| Attribute | Setter  (returns 0 or error code) | Getter  (returns pthread object) |
| Schedule policy | pthread\_attr\_setschedpolicy() | pthread\_attr\_getschedpolicy() |
| Stack address | pthread\_attr\_setstackaddr() | pthread\_attr\_getstackaddr() |
| Stack size (bytes) | pthread\_attr\_getstacksize() | pthread\_attr\_setstacksize() |
| Attribute inheritance (y/n) | pthread\_attr\_setinheritsched() | pthread\_attr\_getinheritsched() |
| Detached state (detach/join) | pthread\_attr\_getdetachstate() | pthread\_attr\_setdetachstate() |
| Scope (process/system) | pthread\_attr\_getscope() | pthread\_attr\_setscope() |
| Scheduling parameters | pthread\_attr\_getschedparam() | pthread\_attr\_setschedparam() |

In our case the thread ittributes are initialize with pthread\_attr\_init() and then the minimum RAM space is allocated to the thread stack:

pthread\_attr\_init(&attr);

pthread\_attr\_setstacksize(&attr, PTHREAD\_STACK\_MIN);

The last attribute is the Scheduling parameters. These specific parameters are saved in a structure named sched\_param. This structure is defined in <sched.h> as below:

struct sched\_param {

int32\_t sched\_priority;

int32\_t sched\_curpriority;

union {

int32\_t reserved[8];

struct {

int32\_t \_\_ss\_low\_priority;

int32\_t \_\_ss\_max\_repl;

struct timespec \_\_ss\_repl\_period;

struct timespec \_\_ss\_init\_budget;

} \_\_ss;

} \_\_ss\_un;

}

The most important member of this struct is the sched\_priority field. This field establishes the priority level of the current thread toward the other ones. Its range is between 1 (the lowest priority level) to 99 (the highest priority level). For example, let’s set this field to 10.

param.sched\_priority = 10;

Then our sched\_param instance must be associated to the pthread attributes thanks to pthread\_attr\_setschedparam():

pthread\_attr\_setschedparam(&attr, &param);

Finally, you will notice the same attribute instance can be used for several threads. Similarly, the same scheduling parameter instance can be associated to several attribute instances.

1. **Routine function definition**

As soon as the thread is created the routine function is executed. Since the goal of this report his routine function is learning how to develop software with Preempt-RT, I suggest defining a function whose the execution duration is long (around 30 seconds). Moreover, this function should alternate calculation, sleep times and displaying on the standard output in order to notice how the OS scheduler executes it in this new thread.

You can set delays with sleep() (in seconds) and usleep() (in microseconds). In that case, don’t forget including unistd.h.

#include <unistd.h>

The routine function called in the new thread must be a void\* function and its argument as well. This is the prototype:

void \* routine\_function\_name (void \* arguments);

void \*thread\_func(void \*data);

1. **Thread creation**

The thread is created and its routine function immediately executed by pthread\_create().

Its first argument is a pointer of the thread. The thread ID will be saved in this variable.

The 2nd argument is a pointer on the attribute structure instance.

The 3rd and 4th arguments are respectively pointers on function and its arguments. Both of them need to be void-typed.

int pthread\_create(pthread\_t \**thread*,

const pthread\_attr\_t \**attr*,

void \*(\**start\_routine*)(void\*),

void \**arg*);

In our case there is no argument for the function:

pthread\_create(&thread, &attr, thread\_func, NULL);

1. **Thread termination**

By calling pthread\_join() the process waits until the created thread terminates. If it has already terminated, pthread\_join () returns immediately. The 1st argument o pthread\_join() is the thread itself, and the 2nd argument a pointer of pointer where the routine function return value is copied.

pthread\_join(thread, NULL);

void \*myRoutineFunction() {

return (void \*) 25;

}

int main(){

pthread\_t myThreadID;

void \*returnValue;

pthread\_create(&myThreadID, NULL, myRoutineFunction, NULL);

pthread\_join(myThreadID, &returnValue);

printf("%d\n",\*(int\*) returnValue);

return 0;

}

void \* myRoutineFunction (void \*myArgument){

\*((int\*) myArgument) = 25;

return;

}

int main(){

pthread\_t MyThreadID;

void \*status = 0;

int myArgument;

pthread\_create(&MyThreadID, NULL, myRoutineFunction, &myArgument);

pthread\_join(MyThreadID, &status);

printf("%d\n", myArgument);

}

**Compillation**

The C files can be basically compiled with GCC. The only precaution to take consists in adding the -pthread flag:

gcc -pthread -o <executable\_file> <C\_file>

[pkurkdjian@gposimrtc03 ~]$ gcc -pthread -o priority10 priority10.c

**Execution**

Now we can execute the program (priority30). It consists in 2 threads created in the main() function. The 2 threads have 2 different scheduling priority levels (1 and 99, the two extremums) but execute the same routine function.

This program will probably be executed by a modern multicore CPU. That’s why you need to specify you want to execute the 2 threads on the same core. Otherwise the 2 threads would be parallelly executed by 2 different cores. In that case you couldn’t put the focus of the experiment on the scheduler behavior because the 2 threads would be executed almost simultaneously. You can force the CPU to execute the program on 1 core only. You can do that with the command taskset (from util-linux package). You need to specify the executable file name and the number of the core you want to use:

[pkurkdjian@gposimrtc03 ~]$ taskset -c <core\_number, another\_one, …> ./<executable\_file>

taskset -c 1 ./priority10 & taskset -c 1 ./priority30

**Cautions**

According to rt.wiki.kernel.org FAQ, the 4 main precautions to take with Preempt-RT are:

* mlockall() should be call very early in the main() function in order to prevent the scheduler to save the process stack in the hard disk.
* All the threads should be created at startup time of the application, and not dynamically. More generally, dynamic programming should be avoided because memory allocation is a slow and non-determinist operation.
* The system calls generating page faults like fopen() should be avoided. The hard disc operations (reading and writing) should be avoided or done before the real-time operations are launched.

**Statistics**

The program has been executed 20 times with different priority level for each thread.

In the following table you can notice the most prioritized thread executes its routine function 2.5 seconds faster than the other thread. The OS scheduler gave the priority to on of the two thread:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Thread 1 | Thread 2 | Execution |  |  | Thread 1 | Thread 2 | Execution |
|  | Priority = 1 | Priority = 99 | gap |  |  | Priority = 99 | Priority = 1 | gap |
| Test 1 (ms) | 20,572 | 18,126 | 2,446 |  | Test 11 (ms) | 18,124 | 20573 | -2,449 |
| Test 2 (ms) | 20,570 | 18,122 | 2,448 |  | Test 12 (ms) | 18123 | 20,573 | -2,450 |
| Test 3 (ms) | 20,573 | 18,123 | 2,450 |  | Test 13 (ms) | 18123 | 20,571 | -2,448 |
| Test 4 (ms) | 20,570 | 18,121 | 2,449 |  | Test 14 (ms) | 18124 | 20573 | -2,449 |
| Test 5. (ms) | 20,574 | 18,122 | 2,452 |  | Test 15 (ms) | 18,123 | 20569 | -2,446 |
| Test 6 (ms) | 20,571 | 18,124 | 2,447 |  | Test 16 (ms) | 18123 | 20,570 | -2,447 |
| Test 7 (ms) | 20,573 | 18,123 | 2,450 |  | Test 17 (ms) | 18121 | 20567 | -2,446 |
| Test 18 (ms) | 20,570 | 18,125 | 2,445 |  | Test 18 (ms) | 18,124 | 20571 | -2,447 |
| Test 9 (ms) | 20,570 | 18,121 | 2,449 |  | Test 19 (ms) | 18,120 | 20569 | -2,449 |
| Test 10 (ms) | 20,575 | 18,124 | 2,451 |  | Test 20 (ms) | 18125 | 20571 | -2,446 |
|  |  |  |  |  |  |  |  |  |
| Average | 20,572 | 18,123 | 2,449 |  | Average | 18,123 | 20,571 | -2,448 |
| Stand. dev. | 1.9 | 1.7 | 2.2 |  | Stand. dev. | 1.5 | 2.0 | 1.5 |

In the next experiment the usleep() function is called 10 000 times in a thread. The goal is measuring the execution duration with Preempt-RT (average and standard deviation) depending on the usleep() duration and the thread priority level. This testing application is priority50.c:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Priority | 1 | 99 | 1 | 99 | 1 | 99 | 1 | 99 | 1 | 99 |
| Sleep() duration | 1us | | 10us | | 100us | | 1 000 us | | 10 000 us | |
| Average | 2.9 us | 2.9 us | 13.2 us | 13.3 us | 103 us | 104 us | 1 005 us | 1 005 us | 10 005 us | 10 005 us |
| Stand. Deviation | 0.3 us | 0.2 us | 1.3 us | 1.1 us | 0.7 us | 0.6 us | 0.7 us | 0.6 us | 0.9 us | 0.9 us |

Conclusion:

* The average execution duration is always longer than the implemented delay (between 2 and 5 us more). This could be due to the usleep() function itself or because of the time measurement functions just before and after the usleep() call.
* The average execution duration does not depend on the priority level
* The standard deviation of the execution duration can be reduced by increasing the priority level. This tip makes the real time application more determinist.
* The standard deviation does not depends on the program duration.