March 12, 2018

RTOS

Lab 1

...

School: ECE Paris
Class: RTOS
Author: Julia DESMAZES
Andreas ORTALDA
Nicolas VERHELST

Contents

1 Latency test

1.1 Activated policies

We can easily get the list of activated policies on a kernel by using the *chrt* commande with option *-m*, see man page.

Listing 1: chrt man page

```
Other options:
 -a, -all-tasks       operate on all the tasks (threads) for a given pid
 -m, -max             show min and max valid priorities
```

Listing 2: chrt -m result

```
$ sudo chrt -m
SCHED_OTHER min/max priority    : 0/0
SCHED_FIFO min/max priority     : 1/99
SCHED_RR min/max priority       : 1/99
SCHED_BATCH min/max priority    : 0/0
SCHED_IDLE min/max priority     : 0/0
SCHED_DEADLINE min/max priority : 0/0
```

1.2 Supported schedualing policies

1.3 Cyclictest

Cyclictest is a programme included in the rt_test suite to measure the latency of an particular environment. It create a defined number of threads that it wakes up at set intervals and then measures the latency between the moment the thread was expected to wake-up and the actual time the action took place.
Reading the *-h* guide we can learn that *-t* allows us to set the number of threads we wish to create and *-p* sets the priority.

threads *-t* More precisely if no parameter is give the number of threads will be equal to the number of CPU's and the execution of throws threads will be balanced equally[1] between they s different CPU's. As we are currently on a 4 CPU system if we wish to create 4 threads leaving it to the default option would be correct. But if we didn't specify the *-t* option then only 1 thread would have been created.

Listing 3: cyclictest -h

```
-t        --threads          one thread per available processor
-t [NUM] --threads=NUM       number of threads:
without NUM, threads = max_cpus
without -t default = 1
```

priorities *-p* In linux priority ranges are fixed by the scheduling policy used. by default cyclictest runs SCHED_FIFO[2] and it's priorities will range from 0 to 99. Here priorities are reversed with 99 being the highest priority and 0 the lowest.

Listing 4: cyclictest -h

```
-p PRIO  --prio=PRIO         priority of highest prio thread
```

1.4 Test

1.4.1 SHED_OTHER

[1] under the best possible conditions

[2] See patch for fix `https://www.spinics.net/lists/linux-rt-users/msg05449.html`

Listing 5: preempt-rt kernel

```
^Cpi@raspberrypi:~/rt-tests$ sudo ./cyclictest --policy=other -t -n
policy: other/other: loadavg: 0.01 0.03 0.03 1/150 1324

T: 0 ( 1229) P: 0 I:1000 C:1242270 Min:      21 Act:    71 Avg:    69 Max:     2541
T: 1 ( 1230) P: 0 I:1500 C: 828181 Min:      21 Act:    67 Avg:    67 Max:     1991
T: 2 ( 1231) P: 0 I:2000 C: 621135 Min:      21 Act:    68 Avg:    68 Max:     4476
T: 3 ( 1232) P: 0 I:2500 C: 496909 Min:      21 Act:    67 Avg:    69 Max:     3180
```

Listing 6: volontary kernel

```
sudo ./cyclictest -t -n

policy: other/other: loadavg: 0.31 0.09 0.02 1/116 702

T: 0 (  695) P: 0 I:1000 C:1220752 Min:      15 Act:    66 Avg:    66 Max:      528
T: 1 (  696) P: 0 I:1500 C: 813835 Min:      15 Act:    65 Avg:    64 Max:      490
T: 2 (  697) P: 0 I:2000 C: 610376 Min:      16 Act:    66 Avg:    67 Max:     1354
T: 3 (  698) P: 0 I:2500 C: 488301 Min:      15 Act:    64 Avg:    65 Max:      545
```

1.4.2 SHED_FIFO

```
^Cpi@raspberrypi:~/rt-tests sudo ./cyclictest --policy=fifo -t -n -p99
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.06 0.04 0.00 1/120 860

T: 0 (  857) P:99 I:1000 C:  81134 Min:      9 Act:   10 Avg:   11 Max:      64
T: 1 (  858) P:99 I:1500 C:  54089 Min:      9 Act:   11 Avg:   12 Max:     102
T: 2 (  859) P:99 I:2000 C:  40567 Min:      9 Act:   10 Avg:   12 Max:      92
T: 3 (  860) P:99 I:2500 C:  32453 Min:      9 Act:   10 Avg:   12 Max:      48
```

Figure 1: SHED_FIFO volontary

```
^Cpi@raspberrypi:~/rt-tests $ sudo ./cyclictest --policy=fifo -t -n -p99
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 18.08 49.73 33.62 1/152 2259

T: 0 ( 2255) P:99 I:1000 C:  56367 Min:     11 Act:   14 Avg:   14 Max:      77
T: 1 ( 2256) P:99 I:1500 C:  37578 Min:     12 Act:   14 Avg:   14 Max:      63
T: 2 ( 2257) P:99 I:2000 C:  28183 Min:     12 Act:   14 Avg:   14 Max:      68
T: 3 ( 2258) P:99 I:2500 C:  22546 Min:     13 Act:   14 Avg:   14 Max:      59
```

Figure 2: SHED_FIFO preempt-rl

1.4.3 SHED_RR

```
^Cpi@raspberrypi:~/rt-tests $ sudo ./cyclictest --policy=rr -t -n -p99
# /dev/cpu_dma_latency set to 0us
policy: rr: loadavg: 0.06 0.03 0.00 1/120 889

T: 0 (  870) P:99 I:1000 C: 105636 Min:      9 Act:   10 Avg:   12 Max:     103
T: 1 (  871) P:99 I:1500 C:  70424 Min:      9 Act:   14 Avg:   13 Max:      84
T: 2 (  872) P:99 I:2000 C:  52818 Min:      9 Act:   10 Avg:   12 Max:      80
T: 3 (  873) P:99 I:2500 C:  42254 Min:     10 Act:   13 Avg:   12 Max:      70
```

Figure 3: SHED_RR volontary

```
^Cpi@raspberrypi:~/rt-tests $ sudo ./cyclictest --policy=rr -t -n -p99
# /dev/cpu_dma_latency set to 0us
policy: rr: loadavg: 8.01 42.11 31.87 1/153 2272

T: 0 ( 2268) P:99 I:1000 C:  43078 Min:     11 Act:   34 Avg:   14 Max:      65
T: 1 ( 2269) P:99 I:1500 C:  28719 Min:     12 Act:   14 Avg:   14 Max:      56
T: 2 ( 2270) P:99 I:2000 C:  21539 Min:     12 Act:   15 Avg:   14 Max:      66
T: 3 ( 2271) P:99 I:2500 C:  17231 Min:     12 Act:   16 Avg:   14 Max:      61
```

Figure 4: SHED_RR preempt-rl

1.4.4 Conclusion

We observe that throughout all the different schedules have about similar avenged execution times with a slight advantage for the voluntary kernel. But, we there is a much higher maximum times for the voluntary kernel than the pre-emptive kernel, this is because the pre-emptive kernel was designed for consistency and to give a best worst case latency by minimizing the part of the kernel that can not be pre-empted. This has negative effects on the advantage execution time that is worst than in the volontary kernel [3] as more overhead is necessary to ensure this determinism behaviour. This involves mechanisms such as priority inheritance to avoiding deadlocks or real-time throttling.

1.5 Cyclictest + Hackbench

1.5.1 Hackbench under SHED_OTHER and priority 20

```
pi@raspberrypi:~/rt-tests $ ./hackbench -l 1000000
Running in process mode with 10 groups using 40 file descriptors each (== 400 ta
sks)
Each sender will pass 1000000 messages of 100 bytes
```

Figure 5: Hackbench under SHED_OTHER and priority 20

1.5.2 Volontary

```
^Cpi@raspberrypi:~/rt-tests $ sudo ./cyclictest --policy=fifo -t -n -p99
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 72.10 34.41 13.61 50/521 2118

T: 0 ( 2114) P:99 I:1000 C:  66022 Min:      7 Act:   11 Avg:   13 Max:     192
T: 1 ( 2115) P:99 I:1500 C:  44015 Min:      9 Act:   12 Avg:   16 Max:     173
T: 2 ( 2116) P:99 I:2000 C:  33011 Min:      9 Act:   20 Avg:   18 Max:     182
T: 3 ( 2117) P:99 I:2500 C:  26409 Min:      8 Act:   17 Avg:   17 Max:     119
```

Figure 6: Volontary fifo with hackbench

```
^Cpi@raspberrypi:~/rt-tests sudo ./cyclictest --policy=rr -t -n -p99
# /dev/cpu_dma_latency set to 0us
policy: rr: loadavg: 82.19 43.52 17.92 108/521 2329

T: 0 ( 2127) P:99 I:1000 C:  43577 Min:      7 Act:   11 Avg:   14 Max:     155
T: 1 ( 2128) P:99 I:1500 C:  29051 Min:      8 Act:   14 Avg:   16 Max:     175
T: 2 ( 2129) P:99 I:2000 C:  21788 Min:      9 Act:   23 Avg:   18 Max:     124
T: 3 ( 2130) P:99 I:2500 C:  17430 Min:     10 Act:   68 Avg:   28 Max:     223
```

Figure 7: Volontary rr with hackbench

1.5.3 Preempt-rt

[3] to help improve the situation lazy pre-emption was introduced for SHED_OTHER, see more https://lwn.net/Articles/522144/

Listing 7: Preempt fifo with hackbench

```
pi@raspberrypi:~/rt-tests$ sudo ./cyclictest --policy=fifo -t -n -p99
policy: fifo: loadavg: 130.65 51.20 18.86 85/551 1404

T: 0 ( 1396) P:99 I:1000 C: 106248 Min:       7 Act:    26 Avg:    20 Max:
78
T: 1 ( 1397) P:99 I:1500 C:  70832 Min:       7 Act:    18 Avg:    20 Max:
73
T: 2 ( 1398) P:99 I:2000 C:  53124 Min:       7 Act:    24 Avg:    20 Max:
73
T: 3 ( 1399) P:99 I:2500 C:  42499 Min:       9 Act:    22 Avg:    21 Max:
74
```

Listing 8: Preempt rr with hackbench

```
pi@raspberrypi:~/rt-tests$ sudo ./cyclictest --policy=rr -t -n -p99
policy: rr: loadavg: 133.77 95.30 43.45 100/551 1454

T: 0 ( 1450) P:99 I:1000 C:  70877 Min:       8 Act:    20 Avg:    20 Max:
91
T: 1 ( 1451) P:99 I:1500 C:  47251 Min:       8 Act:    20 Avg:    21 Max:
83
T: 2 ( 1452) P:99 I:2000 C:  35438 Min:       9 Act:    23 Avg:    21 Max:
69
T: 3 ( 1453) P:99 I:2500 C:  28350 Min:       9 Act:    21 Avg:    21 Max:
92
```

1.5.4 Hackbench under RT scheduling policies and priority 49

1.5.5 SHED_FIFO

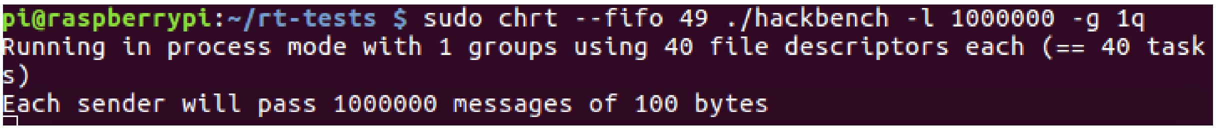


Figure 8: Hackbench with shed_fifo and priority 49

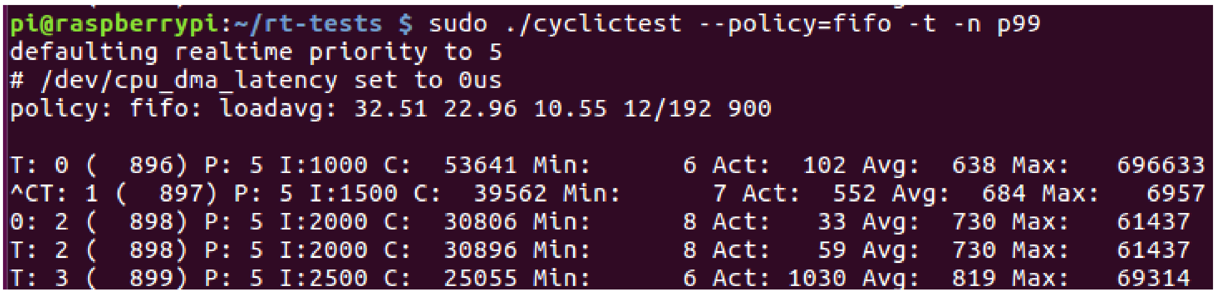


Figure 9: Volontary with fifo

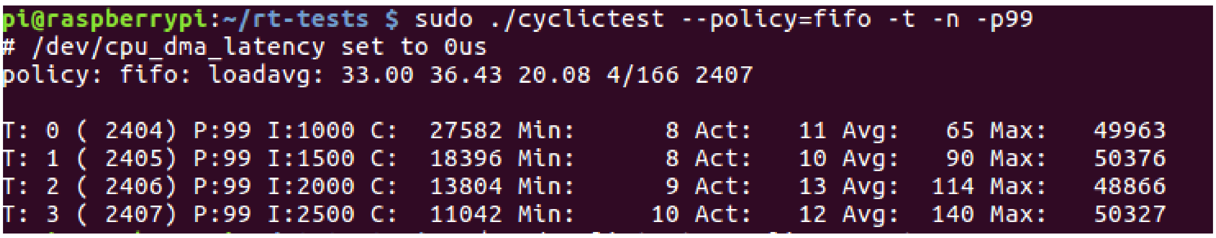
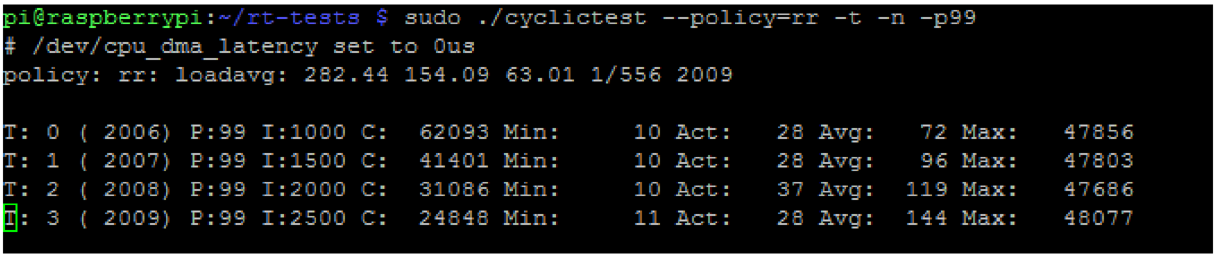


Figure 10: Preempt-rt with fifo

1.5.6 SHED_RR

Listing 9: Hackbench with shed_rr and priority 49

```
pi@raspberrypi:~/rt-tests$ \textsc{sudo chrt --rr 49 ./hackbench -l 1000000 -g 1q
}
Each sender will pass 1000000 messages of 100 bytes
```



```
pi@raspberrypi:~/rt-tests $ sudo ./cyclictest --policy=rr -t -n -p99
# /dev/cpu_dma_latency set to 0us
policy: rr: loadavg: 282.44 154.09 63.01 1/556 2009

T: 0 ( 2006) P:99 I:1000 C:  62093 Min:     10 Act:   28 Avg:   72 Max:   47856
T: 1 ( 2007) P:99 I:1500 C:  41401 Min:     10 Act:   28 Avg:   96 Max:   47803
T: 2 ( 2008) P:99 I:2000 C:  31086 Min:     10 Act:   37 Avg:  119 Max:   47686
T: 3 ( 2009) P:99 I:2500 C:  24848 Min:     11 Act:   28 Avg:  144 Max:   48077
```

Figure 11: Preempt-rt with fifo

Listing 10: Volontary kernel

```
sudo ./cyclictest --policy=rr -t -n -p99
policy: rr: loadavg: 29.39 27.96 14.37 32/161 848

T: 0 (  832) P:99 I:1000 C: 338191 Min:      8 Act:   18 Avg:   67 Max:   49660
T: 1 (  833) P:99 I:1500 C: 225480 Min:      8 Act:   54 Avg:   96 Max:   49563
T: 2 (  834) P:99 I:2000 C: 169312 Min:      8 Act:   18 Avg:  118 Max:   49511
T: 3 (  835) P:99 I:2500 C: 135316 Min:      9 Act:   19 Avg:  144 Max:   47964
```

1.5.7 Consclusion

With hackbench we are introducing significant stress on our systems and are truly revealing the core difference between the voluntary and the full preemptable kernel : consistency. Avenged times are grouped more closely and although minimum times are vagly similar the interval between minimum and maximum times are vastly different in favour of the pre-emptable kernel. This is particularly apparent when hackbench is given a priority of 49 and using fifo. As our task have a higher priority we are pre-empting the hackbench tasks and running the cyclictest instead in the case of the preepmt-rt kernel. On the other hand the voluntary kernel is unable to perform they types of pre-emptions: the kernel has no way of interrupting these running routines *eg : a systemcall*. The problem now is that hackbench's principal revolves around systemcalls: the voluntary kernel's tasks will not be able to preempt[4].
Hackbench code :
`https://github.com/linux-test-project/ltp/blob/master/testcases/kernel/sched/cfs-scheduler/hackbench.c`

```
        for (i = 0; i < loops; i++) {
                for (j = 0; j < ctx->num_fds; j++) {
                        int ret, done = 0;

again:
                        ret =
                            write(ctx->out_fds[j], data + done,
                                  sizeof(data) - done);
                        if (ret < 0)
                                barf("SENDER: write");
                        done += ret;
                        if (done < sizeof(data))
                                goto again;
                }
}
```

Figure 12: Hackbench : Sender code

[4] It's dead jim.

```
        /* Receive them all */
        for (i = 0; i < ctx->num_packets; i++) {
                char data[DATASIZE];
                int ret, done = 0;

again:
                ret = read(ctx->in_fds[0], data + done, DATASIZE - done);
                if (ret < 0)
                        barf("SERVER: read");
                done += ret;
                if (done < DATASIZE)
                        goto again;
        }
```

Figure 13: Hackbench : reciver code

2 ADA concurrent programs

2.1 Program 1

```
with Ada.Text_IO;
use Ada.Text_IO;
procedure Main is
--decleration of tasks
   task A;
   task B;
   task body A is
begin
loop
Put_Line("A");
Put_Line("B");
delay 1.0;
end loop;
end A;
   task body B is
begin
loop
Put_Line("C");
Put_Line("D");
delay 1.0;
end loop;
end B;
--creation of 2 instances of task
--main task
begin
null;
end Main;
```

Figure 14: Program 1

code

Observations The A,B,C,D are printed in order throughout the execution with there not being any apparent modification of that ordered sequence.

2.2 Program 2

```
with Ada.Text_IO;
use Ada.Text_IO;
with Ada.Real_Time;

with Ada.Calendar;


procedure Main2 is
--decleration of tasks
   task type A;
   task type B;
   task body A is
use Ada.Calendar;
begin
loop
delay 0.1 ;
Put_Line("A");
Put_Line("B");

end loop;
end A;
   task body B is
--declare usage
use Ada.Real_Time;
--variables of task
Wait_Time: Ada.Real_Time.Time;
Interval :constant Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds (100);
begin
loop
Wait_Time := Clock+Interval;
delay until Wait_Time ;
Put_Line("C");
Put_Line("D");

end loop;
end B;
--creation of 2 instances of tasknatm

type A_ptr is access A;
type B_ptr is access B;

AA : A_ptr;
BB : B_ptr;
--main task
begin
AA := new A;
BB := new B;
end Main2;
```

Figure 15: Program 2

code

Observations In this test we can observe that even though both treads are configure to sleep for the same 100ms time interval they eventually end up off sync. We can attribute this to the difference between the way time is treated in the *delay* and *delay until*.

delay This main difference is that *delay* uses an *approximate relative time delay.* It measured the time from a specific reference point . If this reference point is displaced in time by the process being pre-empted for example this introduces a delay of *at least* the amount of time specified,not the exact specified task. This introduced local drift (delay) can be cumulated resulting in this de-synchronization. If we wanted delay to wait for an absolute time then the task would have to run uninterrupted by any other task which is not the case here.

delay until This statement schedules for an absolute wake-up task, this works by making the task not schedule until the interval time has elapsed. Her again there is no guaranty on the actual time the task will be executed if for example the resource required to handle that task is not available (used by a higher priority task , ect ...) the task will still eventually stall. Yet, as there is no resource sharing in this case we can amuse that the task will be handled immediately[5] after it is marked as ready.

[5]With very little latency