Advanced OS Exam

Lorenzo Landolfi

July 27, 2018

1 Introduction

This project presents an experiment to measure kernel latency on a modern linux system. The aim of the exercise is to analyze the impact of different kernel preemption models on kernel latency, especially during periods of heavy computational or I/O loads.

2 Theory

The kernel *Latency* L of a job J is defined as the difference between the *theoretical* schedule time t and its *actual* schedule time t'. Hence

$$L(J) = t' - t \tag{1}$$

The latency L is composed by three main components $L = L^1 + L^2 + L^3$

1. L^1: caused by delayed interrupt generation
2. L^2: called *non-preemptable section latency*, also L^{np}
3. L^3: called *scheduler latency*

The latency L^1 is caused by hardware issues, it is generally smaller than L^2, but not when the device generating the interrupt is a timer. In this case it is bounded by the interrupt generation

period T^{tick}.

Such a latency is experienced by all the tasks that are willing to sleep for a certain amount of time. For example the actual wake up time t' for a job that has to wake up at time r is $t' = \lceil \frac{r}{T^{tick}} \rceil T^{tick}$.

The latency L^2 is the delay between when an event is generated and when the kernel handles it. It is basically due to non-preemptable sections in the kernel.

L^3 is not really a latency because it is caused by the execution tasks with higher priority and it is taken into account by the scheduling algorithms.

L^2 is in turn composed by three addends

- Interrupt disabling
- Delayed interrupt service
- Delayed scheduler invocation

Interrupt disabling is just the time to execute a protected instruction by the kernel (STI or CLI on x86).

Delayed interrupt service is caused by additional work done by the kernel after the Interrupt Service Routine (ISR) started.

Delayed scheduler invocation happens when there is a delay in the invocation of the scheduler. Such delay is due to the attempt of the kernel of minimizing the number of context switches.

L^{np} is in general entirely dependent on how the kernel handles mutual exclusion on its internal data structures, interrupts and parallelism.

3 Experiment

The experiment was performed on an Ubuntu 16.04 machine equipped with an Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz (6 cores, 12 SMT), kernel version 4.16.12. Latency measurements

are obtained thank to Cyclitest, a tool that accurately and repeatedly measures the difference between a thred's intended wake up time and the time at which it actually wakes up. Cyclictest works as follows. A master thread (SCHED_OTHER) starts a user defined number of threads with a defined real-time priority (SCHED_FIFO). The slave threads are woken up periodically with a defined period and the difference between the scheduled and the actual wake pu time is calculated by the master thread via shared memory.
The plots reported in this document are generated by a script that starts cyclictest with a single measuring thread woken up every 100 μs and then triggers several scripts to stress the system. The scripts generating load on the system are the following.

- CPUload.sh: generates CPU-I/O load by spawning processes executing dd ad kill
- duload.sh: generates CPU-I/O load by spawning processes executing du
- ioload.sh: generates CPU-I/O load by spawning processes executing tar
- networkload.sh: generates CPU-I/O load by spawning processes executing nc
- pingload.sh: generates CPU-I/O load by spawning processes executing ping

The script launches one of the aforementioned executable one after the other stopping the previous one every 25 seconds.
Figures 6, 7, 8, 9 show the results of the experiments with plots ant tables. The plots show the estimated job latency in μs during the cyclictest execution time, whilst the tables contain the maximum, average and standard deviation of the latency got during stress conditions, measured in micro seconds.

CPUload.sh, Figure 1, launches some background processes that copy 1024 blocks of 1GB from /dev/zero to an output file and then some processes that try to kill all the processes having a specified name.

```
while true; do dd if=/dev/zero of=bigfile bs=1024000 count=1024;
   done &
while true; do killall hackbench; sleep 5; done &
while true; do hackbench 20; done &
while true; do dd if=/dev/zero of=bigfile bs=1024000 count=1024;
   done &
while true; do killall hackbench; sleep 5; done &
while true; do hackbench 20; done &
while true; do dd if=/dev/zero of=bigfile bs=1024000 count=1024;
   done &
while true; do killall hackbench; sleep 5; done &
while true; do hackbench 20; done &
```

Figure 1: CPUload.sh script

duload.sh, Figure 2 spawns several processes that simply scan all the file system.

ioload.sh, Figure 3, launches some background processes that compress with the tar command a big directory system containing thousands of files.

networkload.sh, Figure 4, launches some background processes that send and receive 100 thousands packets of 10 MB size.

pingload.sh, Figure 5, launches some background processes that pinging localhost continuously.

4 Discussion

The experiment described in Section 3 highlights two main facts.

1. Latency of non real time systems is massively influenced by the duload.sh script
2. Only the real time kernel manages to bound the maximum kernel latency effectively.

The highest latency peak (6.3 ms) is found during the duload.sh execution in the Vanilla kernel experiment, reported in Figure 6. The duload.sh is without any doubt the script that stresses non real time systems the most. In fact the highest value of latency happens during its execution in all the experiments run on non real time systems (Figures 6, 7, 8).

```
while true; do taskset -c 0 du / ; done &
while true; do taskset -c 1 du / ; done &
while true; do taskset -c 2 du / ; done &
while true; do taskset -c 3 du / ; done &
while true; do taskset -c 4 du / ; done &
while true; do taskset -c 5 du / ; done &
while true; do taskset -c 6 du / ; done &
while true; do taskset -c 7 du / ; done &
while true; do taskset -c 8 du / ; done &
while true; do taskset -c 9 du / ; done &
while true; do taskset -c 10 du / ; done &
while true; do taskset -c 11 du / ; done &
```

Figure 2: duload.sh script

```
while true; do taskset -c 0 tar cvzf test1.tgz ./linux-stable ; done
    &
while true; do taskset -c 1 tar cvzf test2.tgz ./linux-stable ; done
    &
while true; do taskset -c 2 tar cvzf test3.tgz ./linux-stable ; done
    &
while true; do taskset -c 3 tar cvzf test4.tgz ./linux-stable ; done
    &
while true; do taskset -c 4 tar cvzf test5.tgz ./linux-stable ; done
    &
while true; do taskset -c 5 tar cvzf test6.tgz ./linux-stable ; done
    &
while true; do taskset -c 6 tar cvzf test7.tgz ./linux-stable ; done
    &
while true; do taskset -c 7 tar cvzf test8.tgz ./linux-stable ; done
    &
while true; do taskset -c 8 tar cvzf test9.tgz ./linux-stable ; done
    &
while true; do taskset -c 9 tar cvzf test10.tgz ./linux-stable ;
   done &
while true; do taskset -c 10 tar cvzf test11.tgz ./linux-stable ;
   done &
while true; do taskset -c 11 tar cvzf test12.tgz ./linux-stable ;
   done &
```

Figure 3: ioload.sh script

```
nc -vvlnp 12345 >/dev/null & dd if=/dev/zero bs=10M count=100K | nc
   -vvn 127.0.0.1 12345 &
nc -vvlnp 12346 >/dev/null & dd if=/dev/zero bs=10M count=100K | nc
   -vvn 127.0.0.1 12346 &
nc -vvlnp 12347 >/dev/null & dd if=/dev/zero bs=10M count=100K | nc
   -vvn 127.0.0.1 12347 &
nc -vvlnp 12348 >/dev/null & dd if=/dev/zero bs=10M count=100K | nc
   -vvn 127.0.0.1 12348 &
nc -vvlnp 12349 >/dev/null & dd if=/dev/zero bs=10M count=100K | nc
   -vvn 127.0.0.1 12349 &
nc -vvlnp 12350 >/dev/null & dd if=/dev/zero bs=10M count=100K | nc
   -vvn 127.0.0.1 12350 &
nc -vvlnp 12351 >/dev/null & dd if=/dev/zero bs=10M count=100K | nc
   -vvn 127.0.0.1 12351 &
nc -vvlnp 12352 >/dev/null & dd if=/dev/zero bs=10M count=100K | nc
   -vvn 127.0.0.1 12352 &
nc -vvlnp 12353 >/dev/null & dd if=/dev/zero bs=10M count=100K | nc
   -vvn 127.0.0.1 12353 &
nc -vvlnp 12354 >/dev/null & dd if=/dev/zero bs=10M count=100K | nc
   -vvn 127.0.0.1 12354 &
nc -vvlnp 12355 >/dev/null & dd if=/dev/zero bs=10M count=100K | nc
   -vvn 127.0.0.1 12355 &
nc -vvlnp 12356 >/dev/null & dd if=/dev/zero bs=10M count=100K | nc
   -vvn 127.0.0.1 12356 &
```

Figure 4: networkload.sh script

```
taskset -c 0 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 1 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 2 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 3 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 4 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 5 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 6 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 7 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 8 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 9 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 10 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 11 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 10 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 11 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 10 /bin/ping -l 100000 -q -s 10 -f localhost &
taskset -c 11 /bin/ping -l 100000 -q -s 10 -f localhost
```

Figure 5: pingload.sh script

The pingload.sh script is instead the one that causes the highest average latency time in all the experiments run on non real time kernels.

As expected, the best average value of the maximum latency found in the experiments is obtained by the full real time kernel, whilst the worst is found in the vanilla kernel. It is interesting to notice that the fully preemptable kernel is in general able to reduce the latency much better than the VKP and vanilla. Anyway the remarkable peak during duload.sh confirms that it is not as reliable as the real time kernel.

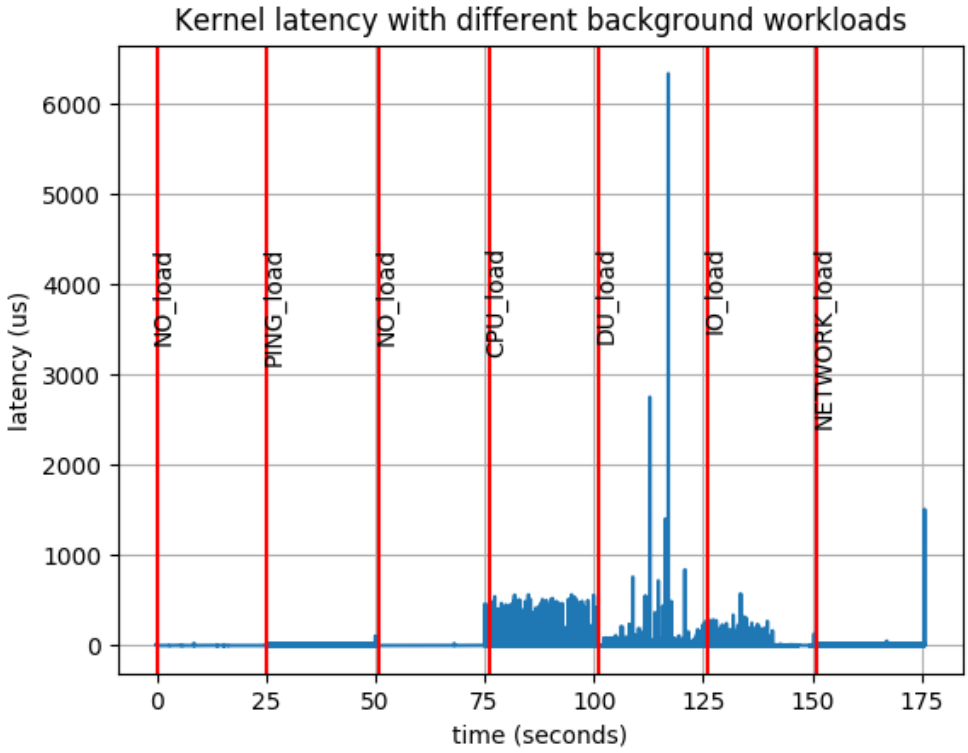
5 Tutorial

This section explains how to run the experiments and how the get the needed tools.

Cyclictest can be compiled from source following the instructions found in

https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rt-tests

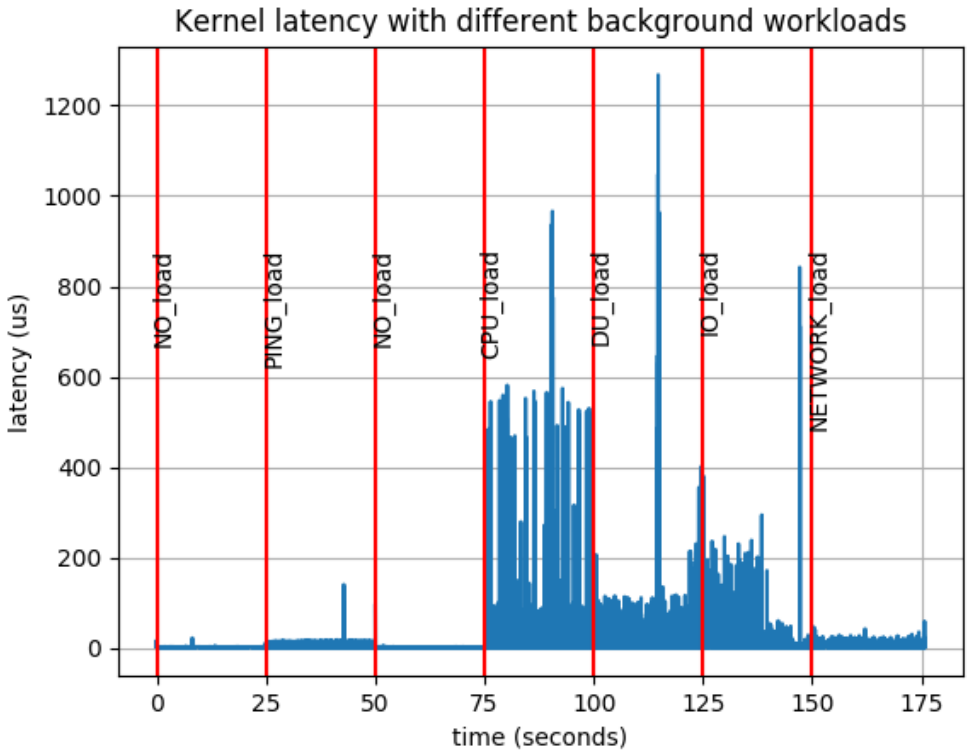
```
sudo apt-get install build-essential libnuma-dev
git clone git://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git
```



	Max	Avg	Std dev
IDLE	28	1.024	0.177
PING	105	**6.654**	3.744
IDLE	462	1.097	1.830
DD	563	2.889	8.549
DU	**6334**	2.501	**15.298**
IO	575	2.953	5.056
NET	1509	3.302	3.453

Average maximum latency: 1368 μ seconds

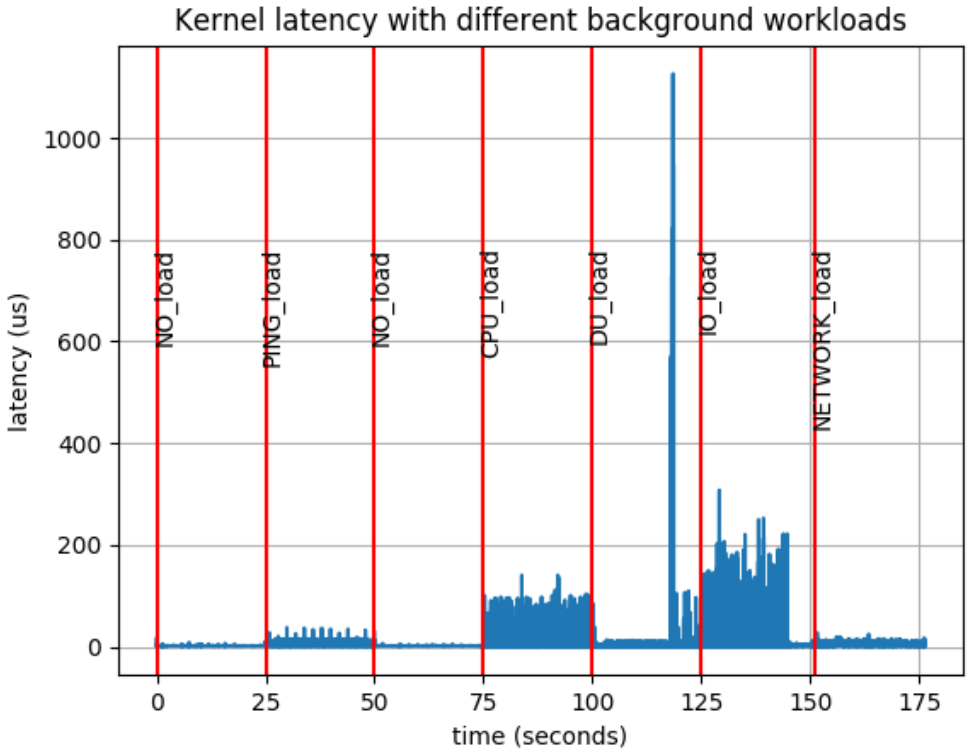
Figure 6: Latency measurement obtained with a Vanilla Linux kernel



	Max	Avg	Std dev
IDLE	22	1.197	0.406
PING	140	**6.735**	3.227
IDLE	97	1.237	0.531
DD	966	3.584	**9.018**
DU	**1268**	3.065	8.759
IO	842	3.318	7.901
NET	59	3.494	1.461

Average maximum latency: 485 μ seconds

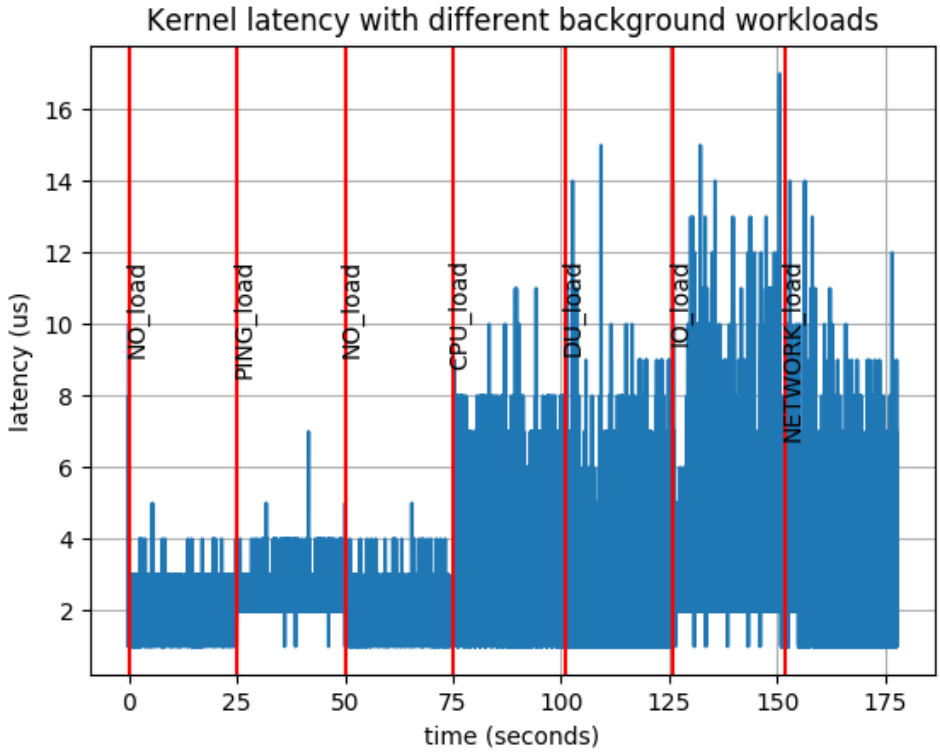
Figure 7: Latency measurement obtained with a Linux kernel compiled with Voluntary Kernel Preemption



	Max	Avg	Std dev
IDLE	18	1.986	0.135
PING	38	**6.341**	3.203
IDLE	14	2.003	0.222
DD	141	3.146	2.403
DU	**1126**	2.499	**5.460**
IO	308	3.319	3.402
NET	28	3.387	0.925

Average maximum latency: 239 μ seconds

Figure 8: Latency measurement obtained with a Linux kernel compiled with Fully Preemption



	Max	Avg	Std dev
IDLE	10	1.963	0.255
PING	7	2.046	0.211
IDLE	5	1.951	0.233
DD	11	2.800	0.731
DU	15	2.353	0.670
IO	**17**	**3.465**	0.688
NET	14	2.813	**0.823**

Average maximum latency: 11.3 μ seconds

Figure 9: Latency measurement obtained with a Linux kernel compiled with Real time

```
cd rt-tests
git checkout stable/v1.0
make all #OR make cyclictest to build only cyclictest
make install
```

The scripts needed run the experiments can be found at https://github.com/llandolfi/AOS. To produce the file reporting all the latencies measured by cyclictest do:

```
sudo su
source ./testintime $interval$ $latencies$ $events$
```

$interval$ is the amount of time each of the scripts generating system load is kept running, $latencies$ is the output file where the latency measurements recorded by cyclictest are written, $events$ is the output file in which the relative starting time of each script generating load is written.

To produce the plots there use plot.py.

```
python plot.py $latencies$ $events$ $period$ $outputimage$
```

In this case $latencies$ and $events$ are the output files produced by testintime.sh, $period$ is the cyclictest period of wake up, $outputimage$ is the filepath in which the produced plot is saved. Finally, in order to produce the values reported in the tables associated to each experiment, it is possible to run the following command.

```
python hist.py \$latencies\$ \$events\$ \$period\$
```

The parameters of this script are the same of plot.py