Lab1 Report

20180472 이승현

Implementation Report

Lab 1 required one to implement a single-core list-based lottery scheduler on OSv. Since this type of scheduler is a simple one requiring no huge changes from the original scheduler mechanism based on weighted runtime, I chose to modify the core scheduling parts from the original code.

The implementation is based on OSv v0.55.0, and thus the diff file must be applied on the same version. Diff file can be applied with `patch -p1 < lab1_diff.txt` command on the OSv directory.

Notable patches are on `core/sched.cc` and `include/osv/sched.hh` with the lottery scheduler implementation and the required APIs. At `cpu::reschedule_from_interrupt` where the original runtime-based scheduling was done, the relevant code was replaced with that of an almost one-to-one matchable lottery scheduler code. Since only a list of runnable threads are needed, `runqueue` is now used as a simple list instead of a runtime-based priority queue. The scheduler simply sums all tickets inside `runqueue`, picks a random number from [0, ticket_sum - 1] using `std::mt19937_64` as PRNG source and `std::uniform_int_distribution` to sample from the range, then iterates through the `runqueue` to find the selected thread. Then we switch to the selected thread unless current running thread equals next thread to run, where in this case we simply reset preemption timer and continue. These cases match almost one-to-one to that of the original implementation, and it's only the scheduling policy that have changed. All the other functions, whether defunct or not, are just left as is (ex: hysteresis-related calls) unless it contradicts with the lottery scheduler logic.

The API is exposed as the same way as (now defunct) priority. `thread::set_ticket(ticket_t ticket)` sets the ticket value as necessary, and thread's ticket value can be fetched by `thread::ticket()`. Ticket value itself is saved in `thread_runtime _runtime` field inside each `thread` object, and the type representing ticket values are `ticket_t`, which is just a typedef of `u32`.

There are three notable values of tickets: `ticket_idle` (1), `ticket_default` (100), and `ticket_infinity` (maximum value of `u32`) each corresponding to the priority equivalents. As the name suggests, `ticket_idle` is the minimum ticket value allocated for idle threads, set at `cpu::init_idle_thread()`. `ticket_default` is the default ticket value initialized for all threads at thread initialization (`thread::thread`). `ticket_infinity` represents the maximum possible tickets allocatable for a thread. Since its priority-equivalent `priority_infinity` is used at driver/virtio-net.cc `net::net`, I have also set the ticket value appropriately to obtain the same results.

Evaluation

To evaluate the scheduler, a simple multi-threaded program is written as a test case at `tests/misc-scheduler-lottery.cc`. `modules/tests/Makefile` is modified accordingly to include the test case at build. Patches on `scripts/setup.py` and `modules/tests/module.py` are just for my local development environment issues, and won't affect the testing as we won't use the commented out parts for our testing. Test code can be built with `./scripts/build fs=rofs image=tests`, and can be run with `./scripts/run.py -c 1 -e /tests/misc-scheduler-lottery.so`.

As required by lab specification, each runs of `ticket_test()` does the following:

a) Spawn threads
b) Set ticket values as given at parameter
c) All threads are started, where each threads runs a simple factorial computation. The <u>end time of factorial computation</u> is saved in the results.
d) <u>All threads run a busy-waiting loop until all threads complete the factorial computation.</u> This is to ensure that the total ticket number does not change due to completion.

e) Print execution time of each threads (or more specifically, execution time of factorial computation)

The code includes 6 tests, where the first three is case 1~3 of the required cases, 4th one is case 3 with ticket values scaled by x10000, 5th one is case 1 with x2 threads (8 threads with 100 tickets), and 6th one is 8 threads with exponentially scaled tickets 100, 200, 400, …, 12800. The code is roughly based on `tests/misc-scheduler.cc`. Below are the results of the 6 tests.

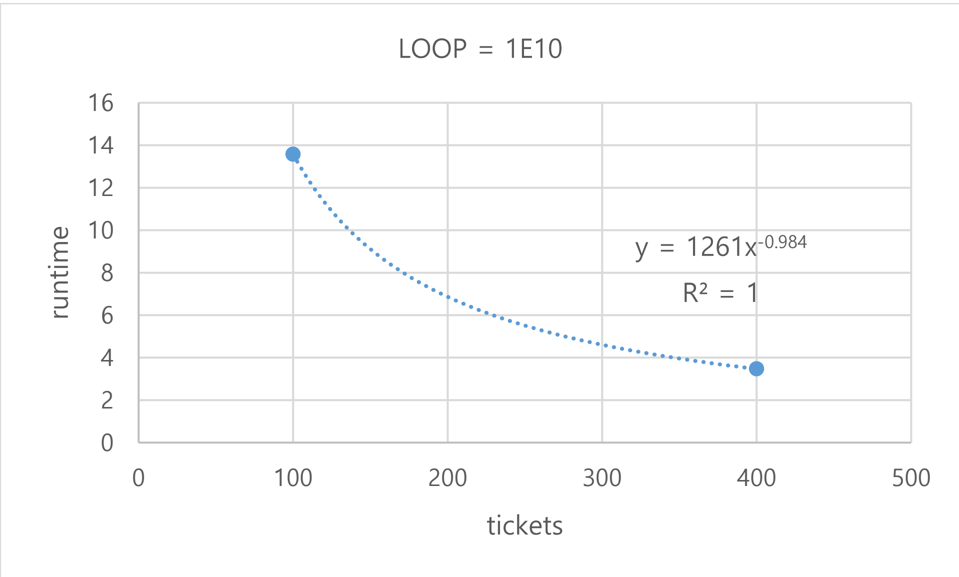
1. {100, 100, 100, 100}

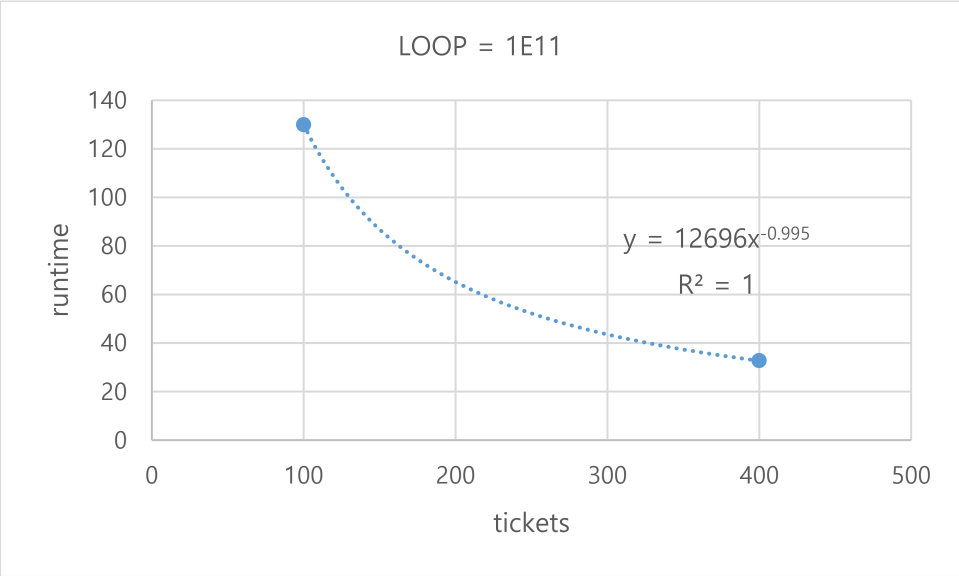
```
[LOOP 1E10]
Ticket test: 100 100 100 100
Ticket #100: 10.0277s (x1)
Ticket #100: 10.1061s (x1.00782)
Ticket #100: 10.4163s (x1.03875)
Ticket #100: 10.9387s (x1.09085)
Ticket test done
[LOOP 1E11]
Ticket test: 100 100 100 100
Ticket #100: 101.759s (x1)
Ticket #100: 103.602s (x1.0181)
Ticket #100: 103.767s (x1.01973)
Ticket #100: 103.881s (x1.02085)
Ticket test done
```

All the four threads for both of the test show approximately the same runtime as expected. Since all the threads' ticket values are equal, we omit the graph plot for this test case.

2. {100, 400}

```
[LOOP 1E10]
Ticket test: 100 400
Ticket #400: 3.47052s (x1)
Ticket #100: 13.5768s (x3.91204)
Ticket test done
[LOOP 1E11]
Ticket test: 100 400
Ticket #400: 32.7035s (x1)
Ticket #100: 129.913s (x3.97246)
Ticket test done
```

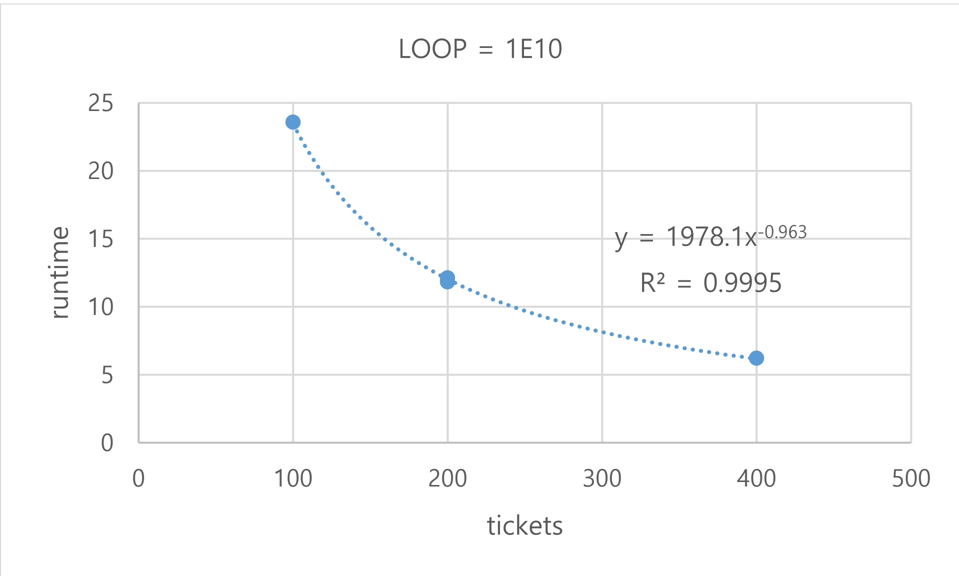


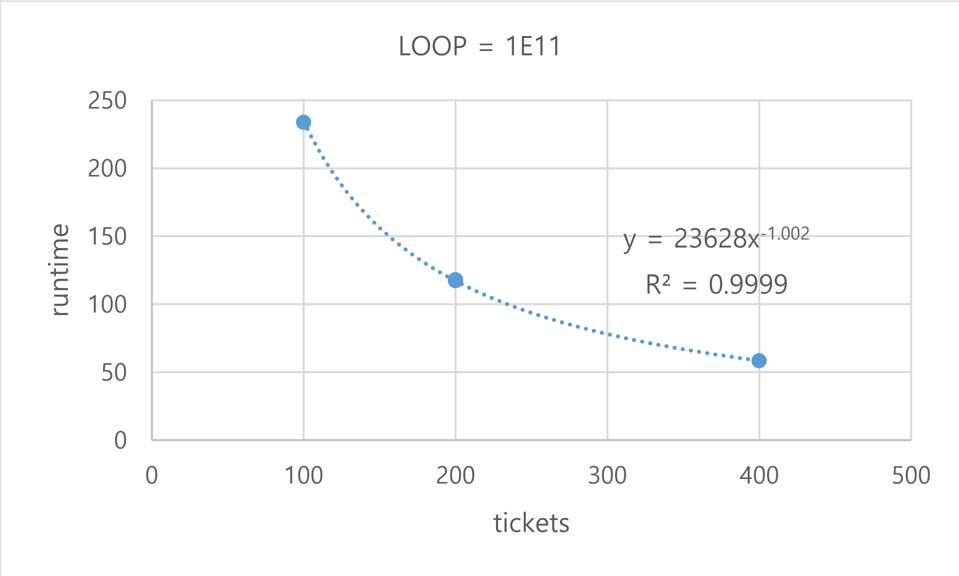


We see that thread with ticket 400 finishes 4 times faster than thread with ticket 100, which shows the inversely proportional relationship. This is because with thread with ticket 400 is 4 times more probable than thread with ticket 100 to be run at each reschedule, causing it to complete 4 times faster.

3. {100, 200, 200, 400}

```
[LOOP 1E10]
Ticket test: 100 200 200 400
Ticket #400: 6.20202s (x1)
Ticket #200: 11.8216s (x1.90609)
Ticket #200: 12.1173s (x1.95377)
Ticket #100: 23.5692s (x3.80025)
Ticket test done
[LOOP 1E11]
Ticket test: 100 200 200 400
Ticket #400: 58.2823s (x1)
Ticket #200: 117.114s (x2.00942)
Ticket #200: 117.901s (x2.02293)
Ticket #100: 233.667s (x4.00922)
Ticket test done
```

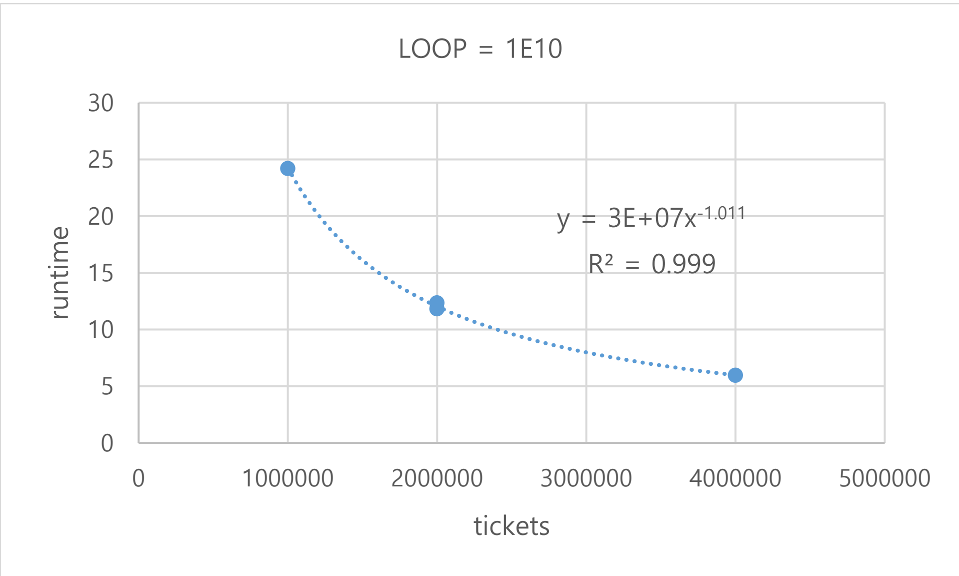


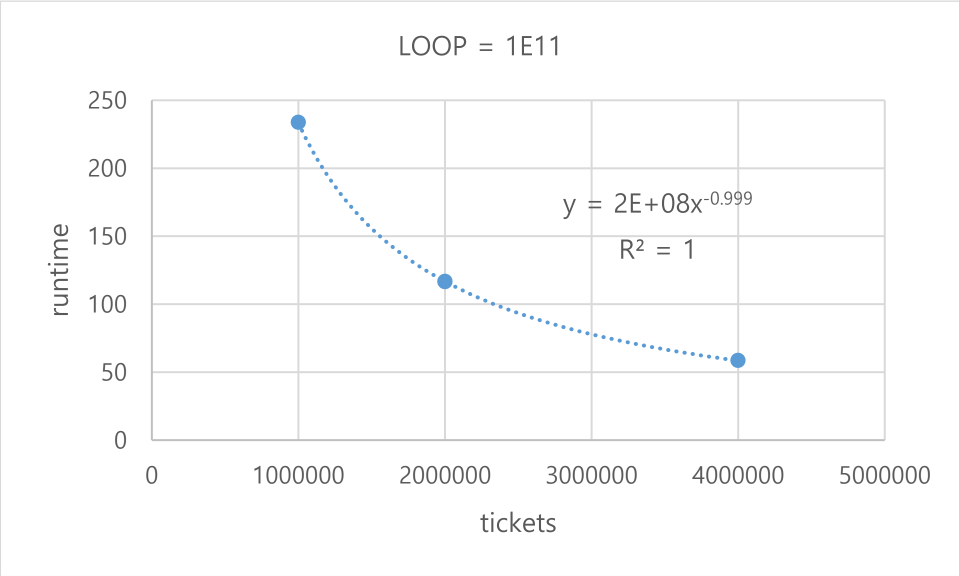


We again see how the runtime is inversely proportional to the tickets (x^a with a ≈ -1, $R^2 \approx 1$)

4. {1000000, 2000000, 2000000, 4000000}

```
[LOOP 1E10]
Ticket test: 1000000 2000000 2000000 4000000
Ticket #4000000: 5.9556s (x1)
Ticket #2000000: 11.8136s (x1.98361)
Ticket #2000000: 12.3362s (x2.07137)
Ticket #1000000: 24.1965s (x4.06281)
Ticket test done
[LOOP 1E11]
Ticket test: 1000000 2000000 2000000 4000000
Ticket #4000000: 58.5345s (x1)
Ticket #2000000: 116.422s (x1.98894)
Ticket #2000000: 116.626s (x1.99242)
Ticket #1000000: 233.656s (x3.99177)
Ticket test done
```





As expected, the results of case 4 is equal to that of case 3 suggesting that there are no wrong dependencies on the scale of ticket sizes, and that the runtime is proportional to the inverse of ticket values. In other words, the portion of CPU time is proportional to ticket values.

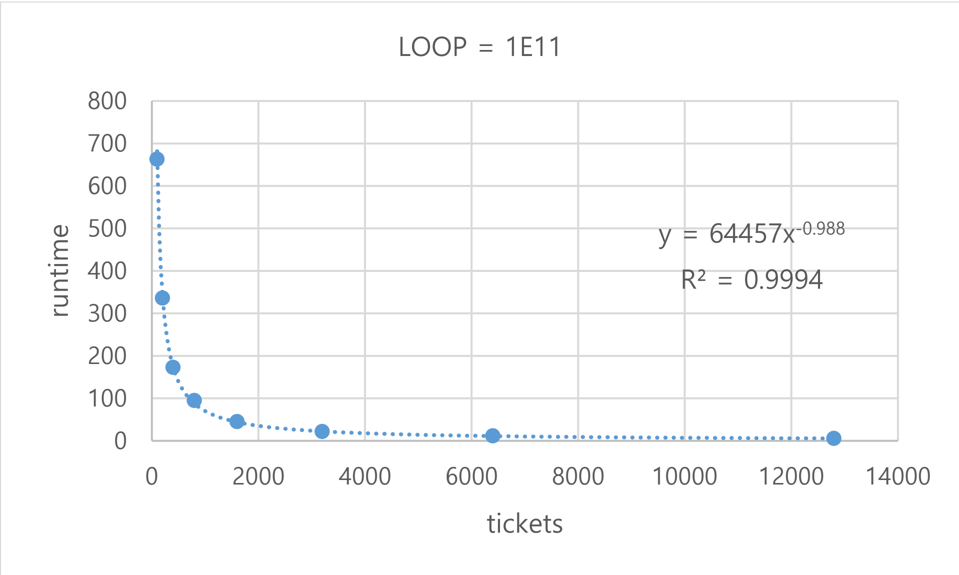
5. {100, 100, 100, 100, 100, 100, 100, 100}

```
[LOOP 1E10]
Ticket test: 100 100 100 100 100 100 100 100
Ticket #100: 23.8757s (x1)
Ticket #100: 24.2099s (x1.014)
Ticket #100: 24.3539s (x1.02003)
Ticket #100: 24.4235s (x1.02294)
Ticket #100: 24.6448s (x1.03221)
Ticket #100: 24.8407s (x1.04042)
Ticket #100: 24.8875s (x1.04238)
Ticket #100: 25.0477s (x1.04909)
Ticket test done
[LOOP 1E11]
Ticket test: 100 100 100 100 100 100 100 100
Ticket #100: 203.189s (x1)
Ticket #100: 204.418s (x1.00605)
Ticket #100: 205.605s (x1.01189)
Ticket #100: 205.612s (x1.01192)
Ticket #100: 206.426s (x1.01593)
Ticket #100: 209.728s (x1.03218)
Ticket #100: 210.767s (x1.0373)
Ticket #100: 211.5s (x1.0409)
Ticket test done
```

As in case 1, all the threads show almost the same runtime. Since we have 2 times more threads to run, the runtime is also 2 times more. Since all the threads' ticket values are equal, we again omit the graph plot for this case.

6. {100, 200, 400, 800, 1600, 3200, 6400, 12800}

```
[LOOP 1E10]
Ticket test: 100 200 400 800 1600 3200 6400 12800
Ticket #12800: 5.39593s (x1)
Ticket #6400: 11.4745s (x2.1265)
Ticket #3200: 21.7206s (x4.02538)
Ticket #1600: 44.9393s (x8.32839)
Ticket #800: 94.5385s (x17.5203)
Ticket #400: 172.722s (x32.0096)
Ticket #200: 335.637s (x62.202)
Ticket #100: 662.73s (x122.82)
Ticket test done
```



For all the tests (excluding case 6 with only LOOP=1E10 tested), we notice that running 1E11 loops generally show less deviation from the theoretical runtime ratio inversely proportional to ticket than 1E10 loops (a is closer to -1, and R^2 value is closer to 1). This is expected since we sample more, and by the law of large numbers the more we run the threads with fixed constant ratio, the more we can expect them to converge at the theoretical ratio. Also, the possible effects of constant overhead due to any reason become less pronounced.