**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 end time of factorial computation is saved in the results.  
 d) All threads run a busy-waiting loop until all threads complete the factorial computation. 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}



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}



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}



We again see how the runtime is inversely proportional to the tickets (xa with a ≈ -1, R2 ≈ 1)

4. {1000000, 2000000, 2000000, 4000000}



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}



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}



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 R2 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.