Load Balancing in OSv's Scheduler

Introduction

Our scheduler keeps a separate run-queue per CPU, ensuring that on each CPU each thread gets its fair share of the CPU, but not guaranteeing that threads which happen to run on different CPU get the same share.

Load-balancing means additional support for migrating threads between CPUs to ensure that all threads get a fair share of the CPUs. For example, if we have 2 cpus and 4 endless-looping threads, two threads stay on one CPU, and two migrate to the second CPU. Moreover, if we have 5 threads on 2 CPUs, we expect a thread (not necessarily the same one) to be continuously moved back and forth between CPUs, so each thread will get the power of exactly ⅖ of a CPU (not ⅓ CPU for 3 threads and ½ for 2 other).

Instantaneous load

When all threads have priority 1, the instantaneous load $l(t)$ in a given CPU is defined as the size of the run queue, plus 1 (unless the running thread is the idle thread, then the instantaneous load is 0).
When threads have different priorities, the instantaneous load is not a straight count of threads, but rather each thread with priority p adds $1/p$ to the load. The rest of this section will be devoted to proving this statement.

Before we can prove anything, we first need to rigorously define the load:

Definition: The instantaneous **load** on a CPU is a function of the CPU fraction that a new default-priority ($p = 1$) thread arriving to this CPU would get, assuming that threads never sleep. If the new thread would get a fraction α of the CPU, the load is defined as $1/\alpha$-1.

This definition coincides with our expectation that a CPU with N runnable threads of default priority will have load N, as a new thread will get $\alpha = 1/(N+1)$ of the CPU, so indeed $1/\alpha - 1 = N$. As another example, if the CPU is idle, the new thread will get all the CPU, so α =1 so that the load is $1/\alpha - 1 = 0$, as expected. As a final example, when we have one thread with $p = 10$, a new default-priority thread will get 10/11 of the CPU, and accordingly the load will be considered to be 0.1.

Lemma: When we have N threads with priorities $p_1, \ldots, p_N$, the share of CPU that thread i gets is:

$$\alpha_i = \frac{1/p_i}{\sum_{j=1}^{N} 1/p_j}$$

Proof: Our scheduler guarantees (up to runtime decay we'll ignore in this discussion) that each of these threads receives the *same* amount of virtual run-time (R) each second. But the virtual runtime accumulation rate is proportional to p, which means that $p_i\alpha_i$ are equal for all i, let's write their value as r. So we have

$$r = p_i\alpha_i \qquad \forall i = 1..N$$

Now, we know that sum of all threads' CPU share is 1, i.e., $\sum_{i=1}^{N}\alpha_i = 1$, so substituting r we get

$$\sum_{i=1}^{N} r/p_i = 1$$

I.e.,

$$r = \frac{1}{\sum_{i=1}^{N} 1/p_i}$$

So finally we get

$$\alpha_i = r/p_i = \frac{1/p_i}{\sum_{j=1}^{N} 1/p_j}$$

Q.E.D.

Corollary: When we have N threads with priorities $p_1, ..., p_N$, the load is $\sum_{i=1}^{N} 1/p_i$

Proof:

If a thread with priority 1 is added, we now have priorities $p_1, ..., p_N,$ and $p_{N+1} = 1$. According to the lemma, the CPU share of the last thread (with the priority 1) is

$$\alpha = (1/p_{N+1})/\sum_{i=1}^{N+1} 1/p_i = 1/(1 + \sum_{i=1}^{N} 1/p_i)$$

So the load, defined as $1/\alpha - 1$, is

$$1/\alpha - 1 = \sum_{i=1}^{N} 1/p_i$$

Q.E.D.

Updating the instantaneous load

We saw about that the instantaneous load is $l = \sum_{i=1}^{N} 1/p_i$, summed on the runnable threads of that CPU. We don't want to calculate this sum on every rescheduling, because the scheduler is currently O(lnN) and we don't want to make it O(N). Instead, we'll need to keep track of the instantaneous load as a separate struct cpu field, and update it every time the run queue is modified (a thread is removed or added), and also every time a thread's priority changes.

Average load

The instantaneous load $l(t)$ is not very useful, because if we see 2 runnable threads at a particular point in time, we don't know if these are actually 2 threads fighting for CPU or just 1

busy thread and a second thread which just woke up and will go back to sleep in a microsecond.

So we want a moving average of recent load. We choose an exponentially decaying moving average with a time constant τ, and define a moving average load $L(t)$ as:

$$L(t_0) = \frac{1}{\tau}\int_0^{t_0} l(t)e^{(t-t_0)/\tau}dt \qquad (1)$$

Note the divisor τ is the integral of the weight function: $\int_0^{\infty} e^{-t/\tau}dt = \tau$, so we indeed have a weighted average. It ensures that if $l(t) \equiv 1$, we'll also get $\lim_{t\to\infty} L(t) = 1$. The factor $\frac{1}{\tau}$ is also evidently necessary just because of dimensional analysis (the integral has units of time, and we want L to be unit-less like l).

Because of our choice of exponential decay, we don't need to remember the entire history of $l(t)$ to calculate L. If we know that between times t_0 and t_1 we had a constant $l(t) = l(t_0)$, we can derive a simple formula (2) for updating $L(t_1)$ from $L(t_0)$:

$$\begin{aligned} L(t_1) &= \frac{1}{\tau}\int_0^{t_1} l(t)e^{(t-t_1)/\tau}dt = \frac{1}{\tau}\int_0^{t_0} l(t)e^{(t-t_1)/\tau}dt + \frac{1}{\tau}\int_{t_0}^{t_1} l(t)e^{(t-t_1)/\tau}dt \\ &= e^{-(t_1-t_0)/\tau}\frac{1}{\tau}\int_0^{t_0} l(t)e^{(t-t_0)/\tau}dt + \frac{1}{\tau}l_0\int_{t_0}^{t_1} e^{(t-t_1)/\tau}dt \\ &= L(t_0)e^{-(t_1-t_0)/\tau} + l(t_0)(1-e^{-(t_1-t_0)/\tau}) \qquad (2) \end{aligned}$$

Updating the average load

Updating the CPU's L needs to be done on every invocation of the scheduler: If t_0 and t_1 are successive invocations of the scheduler, we know that the run-queue had the same length throughout the duration, and it only (potentially) changed right before calling the scheduler. So in formula (2), we need to use for $l(t_0)$ the instantaneous load as it was in the previous scheduling event, not now.

As an example of the importance of taking the instantaneous load after the *previous* scheduling event, not the current one, consider the case when for a very long time the CPU was idle ($l = 0$) and then one thread is woken and reschedule is called. The current instantaneous load is 1, but the load during the entire period was 0, not 1.

Unix load average

BSD Unix and Linux implement the getloadavg() function. This function returns the number of runnable threads throughout the system (total of all CPUs), averaged over 1, 5 and 15 minutes. I think it would be easiest if we calculate these averages using formula (2), but not

starting with the instantaneous load l but rather starting with the short-term (τ) moving average L. Averaging the average will allow us to do this less frequently - e.g., during each load balancing operation and not on every scheduling operation. So the load-balancer thread can update the per-cpu 1,5 and 15-minute moving averages, and a getloadavg() call will simply sum up these values for all CPUs.

Independent, single-thread, load balancing

This is probably the simplest load-balancing scheme to consider, but I believe should already achieve acceptable results. NOTE: It is the same mechanism we already have in the code, just with instantaneous load replaced by average load.

In this scheme: Each CPU runs an independent load-balancing thread. This thread wakes up every once in a while (e.g., once every 100ms) as checks if this CPU has a higher average load (L) than any other CPU's. If it finds a remote CPU with a lower load, it picks one thread with the **highest runtime** on its run-queue and sends it to the remote CPU.

On first glance, there are many problems in this scheme which is why I called it the "simplest. But I believe this is already a useful scheme, because of the following observations:

- One worry about independent load-balancing threads is that they may happen in lock-step: Imagine that CPU A has two threads, CPU B has one. Now CPU A's load balancer moves one thread to CPU B, and a microsecond later CPU B's load balancer runs and potentially returns the thread back to CPU A - and then both load balancers go to sleep for 100ms. Rather than CPU A oscillating between 1 and 2 threads, it has 2 threads most of the time, with 1 thread for only very short durations.
  However, with the moving-average load, this problem cannot actually happen: If CPU A had load 2, and CPU B had load 1, then after the migration, the average load of A is still very close to 2, and that of CPU B is very close to 1, so the migration back will not happen until enough time has passed (order of magnitude of τ) for the loads to reverse their order.
- Continuing the above example, what if the *same* CPU's load balancer runs again after migrating a thread, and the load averages are still close to their original values? It can cause us to to migrate *another* thread off this CPU!
  The solution is to set the load balancer's wait period to not much less than τ, so that between successive runs of the load balancer, the average load has mostly settled down to its new value.
- Consider 3 busy-loop threads running on 2 CPUs. At each instant, we have one CPU with instanatenous load 2, and one with 1, and every once in a while (as explained above) we move one thread from CPU A to CPU B, and later vice-versa.
  It might **seem** that we can't just move the same thread back and forth, because this thread will always see a competitor and thus get 0.5 CPU, while the two other threads

each get 0.75 CPU.
But I believe we're actually fine here: The thread will move together with its runtime, which will be significantly lower than the runtime of the single thread on the destination CPU, so the moving thread will "catch up" at the expense of the "spoiled" thread which until now ran alone.
However, if too much time passes, as the runtime decays we start to lose track of how behind the moving thread is. So we probably need to take the period between successive load-balancer calls (on the same CPU) to be in the ballpart of τ or less. Since we already suggested that it should not be much less than τ, we end up requiring that the load-balancer period should be around τ.

- What if some usually-sleeping thread happens to be awake during the load balancing? Can we accidentally pick it for migration, helping nothing?
  This usually won't happen, because as stated above, we pick for migration the thread with the highest runtime; The usually-sleeping thread will have the lowest, not highest, runtime, so we don't pick it for migration.
- A related problem is that by chance the load-causing threads might be sleeping. We can even see a high load but zero threads in the run queue. This situation should be very rare, and if it happens, we'll just repeat the algorithm again next time.
- What if we start 10 threads on CPU A? It will take 5 iterations of the load balancer until 5 of the threads will move to the second CPU. We can change the code to migrate multiple threads in one iteration, but we believe that in most cases (where the number of runnable threads is small), this will not be a noticeable problem.

The above suggested technique, of migrating the highest-runtime thread, has one significant problem, the **intermittent thread** problem**:** Consider two CPUs, running two busy-loop threads, plus one "intermittent thread" - a thread which runs for 1ms, sleeps for 10ms, and so on ad infinitum. One CPU will run just one busy thread, and get load average of 1.0, and the second CPU will run the busy thread and the intermittent thread, and will get load average of around 1.1. When the load balancer on the second CPU runs, it notices 1.1 > 1.0, so it sends the highest-runtime thread - the busy thread - to the first CPU! Now the first CPU has two busy threads, and the second CPU has just a rarely running thread… Only when the load balancer on the first CPU gets to run, will it move one of the busy threads back to second CPU.
One way to fix the intermittent thread problem is to special-case the idle CPU case: When a CPU becomes idle, it wakes up other load balancers to send it work. In our example, the second CPU, when the intermittent thread goes to sleep, will wake the first CPU's load balancer immediately and have it sent it work.
A different solution is to say that when we have load avarages 1.1 vs 1.0, we don't want to migrate the busy thread (responsible for 1.0 of the load) but rather the intermittent thread (responsible for 0.1 of the load), as this has a better chance of equalizing the load. However, this might always be a good idea. Consider 3 busy threads on 2 CPUs. Good load balancing will require us to send one thread back and forth between CPUs, to achieve average load 1.5

on both CPUs. When the load balancer wakes, it might see, because recently CPU 1 had two threads running and CPU 2 only 1, that CPU 1 has load average, 1.6 and CPU 2 has 1.4. At this point, it is expected to move one of the busy threads from CPU 1 to CPU 2. If there was one intermittent thread causing load 0.1, it would be pointless to move it: The difference between 1.6 and 1.4 is not representative of the *current* load imbalance - it is just a rolling average. The current load imbalance is 2.0 vs. 1.0, and only moving a half-busy thread can solve it (or moving a fully-busy thread back and forth).

TODO: What if two CPUs load-balance at roughly the same time, and both decide to send a thread to a third CPU?
TODO: If load1 = load2+0.001, do we also start migration? Or do we need a more siginicant difference? Which? 1? 0.5? Or what?
TODO: add 3 and 5 thread test cases to tests/misc-loadbalance.so

Some incomplete thoughts:

A thread's share of the load

Consider thread with runtime R living in a CPU with load L. Can we say what we expect will happen to the load if the thread stops running (e.g., migrated to a different CPU)?
This is the thread's share of the load. A busy-loop thread should always have a load share of 1.

The runtime-based load-share measure will never be perfectly correct. Consider one CPU with two threads: One busy-loop thread, and the second a thread which runs for 10ms, sleeps for 10ms, and so on. Our scheduler ensures that each get their fair 50% of the CPU, each will have the same runtime and may appear to have the same load share. And yet, if one thread stops, the load would drop to 1, but if the other thread stopped, the load would drop to 0.5…

Even without solving this question precisely, we can at least avoid migrating a thread with very small runtime (the current code picks a runnable thread randomly, so can pick a thread which happened to run by chance, but usually sleeps). THINK: Maybe it doesn't matter if we move such non-interesting thread. Simply, the load won't drop, and we'll move another thread later.