Requirements:
- global fairness
- cheap to compute

Each task has a measure of runtime it receives from the scheduler. We could use the task's total runtime, t, but we would like to consider only the recent history so that a task that has slept for a long time would not dominate the processor. So we use a decaying average:

$$R(t + \Delta t) = (1 - k\Delta t)R(t) + p(t)r(t)k\Delta t \qquad (1)$$

Where

- R is a measure of the runtime the thread has received in the recent past
- p is the thread's normalized priority[1]
- r describes the runtime history; it has the value 1 when the thread is running, and 0 otherwise
- k is a constant determining how soon the scheduler "forgets" a thread's history, typical value is 50 1/s (equivalent to forgetting history in a few multiples of 20 ms).
- Δt is a small period

Since we don't want to have periodic scheduler ticks, we take the limit $\Delta t \to 0$ and integrate. This yields the normal continuous decaying average function:

$$R(t_0) = \int_0^{t_0} p(t)r(t)e^{k(t-t_0)}dt \qquad (2)$$

The scheduler picks the runnable thread with smallest R, and runs it until some other thread has a smaller R. To prevent excessive context switches, hysteresis is employed.

Computing R continuously is expensive, but not needed. A little algebra shows how to compute $R(t_2)$ given $R(t_1)$, provided p(t) and r(t) have not changed between t1 and t2. This means we can update R only when the scheduler is invoked, or when priorities change, since r(t) only changes as a result of scheduler execution.

$$R(t_2) = e^{k(t_1-t_2)}R(t_1) + \frac{1}{k}p(t_2)r(t_2)(1 - e^{k(t_1-t_2)}) \qquad (3)$$

It is still impractical to compute R for all threads on every scheduler invocation; but we note that, for a given processor, $r(t_2)$ has the value 1 for at most one process; the others are not running (queued or sleeping). The second term of the equation vanishes; and the first is a multiplication by a constant (across all threads, for a given t_2).

[1] This is not the normal unix priority, but rather a value computed from that priority. Here, p should lie in the range (0, 1], with lower values causing the thread to receive more runtime.

Let us introduce an unnormalized runtime measure, R'. This measure is local to a processor; when moving a thread to a different processor, we must normalize it again:

$$R'(t) = c(t)R(t) \qquad (4)$$

If we define

$$c(t_2) = e^{-k(t_1-t_2)}c(t_1) \qquad (5)$$

Then the unnormalized runtime measure is given by

$$R'(t_2) = R'(t_1) + c(t_2)\tfrac{1}{k}p(t_2)r(t_2)(1 - e^{k(t_1-t_2)})$$

Or equivalently (considering (5))

$$R'(t_2) = R'(t_1) + \tfrac{1}{k}p(t_2)r(t_2)\,(c(t_2) - c(t_1)) \qquad (6)$$

For non-running threads, $r(t_2) = 0$, so:

$$R'(t_2) = R'(t_1) \qquad (7)$$

So on each scheduler invocation, we only need to update c and R' for the running thread.

The value of c starts at 1 and increases towards infinity; to avoid overflow of c and R', we need to renormalize R' periodically, by dividing it (for all threads) by the current value of c, and then setting c to 1. This is done rarely enough so that the cost is amortized. See below for more ideas about the renormalization step.

Runnable threads are stored in a sorted container, with R' as the key. When a thread is run, it is taken out of the container, and has its R' updated when it is stopped. When a thread is migrated, R' is normalized first (in the cpu it was running on) and then unnormalized again (in the cpu it is migrated to).

To achieve hysteresis, the scheduler reduces the running thread's R value by a constant t_{gran} when it starts running the thread, and increases it back by the same amount when it stops running. This prevents the thread from being preempted immediately by a thread with the same or similar R value, yet preserves fairness.

When a runnable (but not running) thread (denoted 'q') becomes the one with the lowest R value (among runnable threads), the scheduler computes the t_s in which it would have an R value lower than the running thread (denoted 'r') if it will continue to run at constant priority p_r:

From (3) we have:

$$R_q(t_s) = e^{k(t_0 - t_s)} R_q(t_0)$$
$$R_r(t_s) = e^{k(t_0 - t_s)} R_r(t_0) + \frac{1}{k} p_r (1 - e^{k(t_0 - t_s)})$$

Setting $R_q(t_s) = R_r(t_s)$ and solving for t_s, we obtain

$$t_s - t_0 = \frac{1}{k} ln\left(1 + \frac{k}{p}(R_q(t_0) - R_r(t_0))\right) \qquad (8)$$

Adjusting for the unnormalized R'=Rc value:

$$t_s - t_0 = \frac{1}{k} ln\left(1 + \frac{k}{p\,c(t_0)}(R'_q(t_0) - R'_r(t_0))\right) \qquad (9)$$

The scheduler sets a timer at t_s which is the next preemption point whenever one of the parameters in the equation changes - the runnable thread, or the priority of the running thread.

Simplifying the code, using τ instead of k

TODO: rewrite the above document with this notation, from the start.

We saw for a thread that has been running from t_1 to t_2 with priority p that:

$$c(t_2) = e^{-k(t_1 - t_2)} c(t_1) \qquad (5)$$
$$R'(t_2) = R'(t_1) + \frac{1}{k} p\,(c(t_2) - c(t_1)) \qquad (6)$$
$$t_s - t_0 = \frac{1}{k} ln\left(1 + \frac{k}{p\,c(t_0)}(R'_q(t_0) - R'_r(t_0))\right) \qquad (9)$$

We can simplify the formulas, and the code with the following three changes:

1. Instead of working with k whose units is 1/time, work with $\tau = 1/k$ whose units is time.
2. R, and correspondingly R', have the units of time. We have multiplications by τ throughout the above formulas, which are wasteful and increases the magnitude when τ is large (e.g., measured in nanoseconds). Instead, let's keep a unitless $R'' = R'/\tau$. This also allows us to change τ from time to time (or from CPU to CPU[2]) without the meaning of R'' changing.
3. Use $t_2 - t_1$ in the formulas, instead of $t_1 - t_2$.

With these three changes, the formulas (5), (6) and (9) become the following, which we use in

[2] When a runqueue has a large number of runnable threads, and each gets to run for a fixed time slice (say, 1ms), cycling through all of them will require a long time, and if tau is too low, history would be forgotten and we will lose fairness. So in the future, we may want to make tau a function of the number of runnable threads.

the code (the functions **ran_for** and **time_until**):

$$c(t_2) = e^{(t_2-t_1)/\tau}c(t_1) \qquad (5\tau)$$

$$R''(t_2) - R''(t_1) = p(t_2)(c(t_2) - c(t_1)) \qquad (6\tau)$$

$$t_s - t_0 = \tau ln\left(1 + \frac{1}{p\ c(t_0)}(R''_q(t_0) - R''_r(t_0))\right) \qquad (9\tau)$$

Linear approximation

The formula we saw above for advancing, (5τ) above, involves the exponential function e^x, for the unit-less $x = (t_2 - t_1)/\tau$.

Calculating this function accurately may be slow. The speed of the scheduler's calcuations is especially important when the scheduler runs a very large number of times per second, and in this case, x is relatively small. Can we calculate e^x more quickly for small x?

We know from the Taylor-series expansion of e^x that

$$e^x = 1 + x + \frac{1}{2}x^2 + O(x^3)$$

So for $x < 10^{-3}$, in other words $t_2 - t_1 < \tau/1000$, the error in approximating e^x by the linear function $1 + x$ is less than x^2, i.e., 10^{-6}, which is around the accuracy of single-precision floating point anyway.

So when $t_2 - t_1 < \tau/1000$ we'll replace formula 5τ by the approximate (up to $O(x^2)$):

$$x = (t_2 - t_1)/\tau$$

$$c(t_2) = (1 + x)c(t_1)$$

Hysteresis

TODO: This section is a big mess and needs to be rewritten.

We need an hysteresis mechanism in order to avoid immediate switching between threads when several threads have identical or very close R.

We choose a time constant t_H, which specifies that if we have two competing threads with identical priorities, and if both have identical R, then if one thread is started it will run for t_H until switching to the other thread.

Note that the steady in the above case will become that the two threads alternate each runing $2t_H$. The reason is after the first thread ran for t_H it comes to the same R as the second thread, but now repays its "loan" of t_H so it is now ahead in t_H. When the second thread runs and "loans" t_H, it has around $2t_H$ to reach the first thread.

As explained above, t_H is necessary to prevent excessive context switching when two busy threads compete: we're run each thread in turn for the "desired timeslice", instead of some infinitesimal time slice. But its important mentioning that t_H is necessary even when we already have a high context-switch rates, i.e., threads which work little until sleeping again. In

that case t_H cannot, of course, reduce the context switch rate, but rather reduces the number of expensive timer reprogramming: Consider a workload where we have a million context switches a second (e.g., our context switch benchmark tst-ctxsw.so). If the difference in R is allowed to be tiny, (9) will discover that we need to switch threads in a, say, 10 microseconds, and we'll need to set up a timer 10 microseconds into the future. After we do that, we don't need to set it again until 10 microseconds have elapse, but after they do, we'll set a new timer,
maybe again to 10 microseconds - and end up with an expensive timer-setting operation 100,000 times each second. With t_H = 1ms, the timer will always be set up to be at least 1ms in the future, so we'll never have more than 1,000 timer settings each second.

To achieve this, we reduce R' and c using (5), (6) with $t_2 - t_1 = -t_H$, then run the thread and increase its R' and c using (5),(6), and finally, use (5),(6) again with $t_2 - t_1 = t_H$. To show this is what we want we need to prove that:

1. In the two-thread case above, the thread will indeed run for t_H before switching to the other thread.
2. The extra steps adding $-t_H$ and at the end t_H cancel out - the three steps result in identical c and R' that the middle step alone would have got.

We'll use the formulas with (5τ) and (6τ) with R'' and τ (todo - eventually the whole document should use them). This is called one "**ran_for(time)**" step in our code.

$$c(t_2) = e^{(t_2-t_1)/\tau}c(t_1)$$
$$R''(t_2) = R''(t_1) + pc(t_2)(1 - e^{-(t_2-t_1)/\tau})$$

ran_for

To make this operation reversible (and useful for negative times), we need to use the values at t_1, not t_2. Substituting $c(t_2)$ on the right-hand-side above, we get a new formulation of ran_for which we need to use (**TODO - it seems the following is correct even without the reversable formulation, we don't really have to switch to it.)**:

$$c(t_2) = e^{(t_2-t_1)/\tau}c(t_1)$$
$$R''(t_2) = R''(t_1) + pc(t_1)(e^{(t_2-t_1)/\tau} - 1)$$

ran_for(t2-t1)

To prove <u>both</u> theorems, we prove that the process of ran_for is additive, i.e., that c and R'' after ran_for(d1) and then ran_for(d2) is equivalent to that of ran_for(d1+d2). This will mean that after a ran_for(-tH), a ran_for(tH) will return it to the original R'', and also that doing ran_for(-tH) and then ran_for(slice) and then ran_for(tH) will be equivalent to just ran_for(slice).

<u>Proof</u>

The additivity of these steps must be true given that they were defined in the term of an integral, but let's check that the actual formulas work:

In one d1+d2 step, we get

$$R''(d1 + d2) = R''(0) + pc(0)\left(e^{(d1+d2)/\tau} - 1\right)$$
$$c(d1 + d2) = e^{(d1+d2)/\tau} c(0)$$

In two steps:
After one d1 step we get

$$R''(d1) = R''(0) + pc(0)(e^{d1/\tau} - 1)$$
$$c(d1) = e^{d1/\tau} c(0)$$

Now doing another d2 step above the former d1 step we get:

$$R''(d1 \text{ and then } d2) = R''(d1) + pc(d1)(e^{d2/\tau} - 1)$$
$$= R''(0) + pc(0)(e^{d1/\tau} - 1) + pe^{d1/\tau} c(0)(e^{d2/\tau} - 1)$$
$$= R''(0) + pc(0)(e^{d1/\tau} - 1 + e^{(d1+d2)/\tau} - e^{d1/\tau})$$
$$= R''(0) + pc(0)(e^{(d1+d2)/\tau} - 1)$$
$$c(d1 \text{ and then } d2) = e^{d2/\tau} c(d1) = e^{d2/\tau} e^{d1/\tau} c(0)$$
$$= e^{(d1+d2)/\tau} c(0)$$

TODO: think what happens here if the priority changes in the middle of the thread's run. I think we just need to stop the thread, return its burrowed time, and start again.

Hysteresis vs Tau

If τ isn't large enough compared to t_H, the scheduler forgets history too quickly and the accuracy of its fairness is accordingly diminished.

Consider for example the case of $t_H = 2ms$ and $\tau = 200ms$, and consider two threads, one with priority 1 and one with priority 4. Perfect fairness means first thread will get 4 times more runtime than the second. But what happens in practice is this: When the two threads have roughly the same runtime, and the p=4 thread is run, it gets to run for $t_H = 2ms$. When it completes its time slice, it will have acquired enough runtime to allow the second p=1 thread to run for 4*2ms = 8ms. But 8ms is not negligable compared to $\tau = 200ms$, and while the second thread is running, the first thread's runtime begin to decay, and the second thread ends up running slightly less than 8ms. The end result is that the second thread will get a bit less than 4 times the CPU time of the first. In an actual test, we measured the 3.95. If we double τ to 400ms we measure a fairer ratio 3.98. Halving t_H instead improves fairness even more (I'm not sure why, maybe a coincidence). If we lower τ to 50ms, the ratio of course becomes even less fair, at 3.75.

This phenomenon is, of course, a feature, not a bug. Perfect fairness over time requires

keeping perfect history, while the whole intention of τ is for the history to decay, and start to forget what happened a long time ago.

TODO: Consider if to prevent this problem from becoming worse, we need to lower t_H for high-priority (low p) threads. So a thread with p=1/4 will use not t_H but rather $t_H/4$ so that the time slices of the p=1 thread will not become very large. But this will create a new problem - a high-priority (low p) thread which runs a lot will cause more timers, and two of them will cause more context switches, than we wanted.

Approximations

A small overshoot is allowable (e.g., in formula (9)), so approximations can be used.
The exponential function is easier to approximate in the form of a power-of-two function, so (6) becomes

$$R'(t_2) = R'(t_1) + \frac{1}{k}p(t_2)r(t_2)(2^{-(log_2 e)k(t_1-t_2)} - 1)$$

Renormalization

As explained above, c starts at 1 (we can also start lower, e.g., the floating point with the lowest mantissa) and increases towards infinity; to avoid overflow we need to renormalize R' periodically, by dividing it (for all threads) by the current value of c, and then setting c to 1.

To do this, we need the list of all threads which are assigned to this CPU. This is not just the list of runnable threads on this CPU - it also includes threads which ran on this CPU (and therefore now have runtimes local to this CPU) but have since gone to sleep.

We don't currently have such a list. One option is to maintain such a linked list (add/remove threads to this list on thread migration, creation and deletion). This list might contain a lot of long-sleeping threads, making the renormalization step O(number of threads in the system). This is not expected to be a serious problem, as renormalization is a rare event - c grows as $c(t) = e^{t/\tau}$, so if the range of c is around 10^{37} (cmax/cinitial in our code), c will overflow in around $ln(10^{37}) = 87$ periods of τ - likely to be several seconds for reasonable choices of τ.

An alternative idea - which we currently use in the code - is to only renormalize the runtime of the runnable threads and the running thread. The runtime of sleeping threads will only be fixed when they are next woken up. This makes renormalization O(runnable threads), but adds a few instructions to each wakeup.
For this to work, we use the following ideas:

1. Note that during renormalization we don't strictly need to divide by the current c. Instead, we can choose a constant **maxc** where we do the renormalization when c > maxc, and divide everything by this constant maxc (not by c). This will be just as

effective in avoiding overflows, but will make it trivial to know how much a runtime that missed one renormalization step needs to be divided by.

2. However, a thread might have been sleeping for so long that it missed more than one renormalization step. We need to know that.
3. For this, we keep a per-cpu *runtime renormalization counter,* and also a per-thread counter with the same name.
4. During a renormalization step on a CPU, we increment the CPU's renormalization counter, renormalize (divide by maxc) the threads in the run-queue and the running thread, and for each of these threads, remember the new runtime renormalization counter.
5. When a thread is being woken up on a CPU, before inserting it into the runqueue, we need to know its runtime (R'). So we compare the CPU's and the thread's runtime renormalization counter:
    a. If the thread's renormalization counter is the same as the CPU's, R' saved in the thread is fine as-is.
    b. If the thread's counter is one below the CPU's, we missed a renormalization step, and need to divide the thread's R' by maxc, and update the thread's renormalization counter.
    c. If the thread's counter is two or more below the CPU's, it means the thread spent more than a whole generation asleep, and the renormalizations it missed (at least two) would have brought its R' to zero close to zero. It doesn't matter if we missed 2 renormalizations or 17 - in any case we can just set R' to zero, and update the thread's renormalization counter the current one

    We don't even need to treat a, b, c and c as separate cases. Basically if we set n as the difference the CPU's and the thread's renormalization counter, we we need to divide R' by max^n and then set the thread's counter to equal the CPU's.

The meaning of the constant τ

We can see from (3) that the constant τ (whose units are time) determines the rate in which a long-sleeping thread loses its runtime history, lowering the runtime gap between two threads that existed sometime in the past.

Let's assume we have two threads, both with the same priority ($p = 1$ for simplicity).
Thread A has been running continuously for a very long time, and thread B has been sleeping for a very long time.
Now, B wants to run. How much time will B get to run?
With perfect history, we would let it run for the same time that A ran, and force B to sleep for that long. But the decay helps us make the history a bit more short sighted, and the prevent long-sleeping threads from monopolizing the CPU when they finally decide to run.

After T seconds, we have according to (3),

$$R_A = \tau(1 - e^{-T/\tau})$$
$$R_B = 0$$

When T was a very long time, we approach

$$R_A = \tau$$
$$R_B = 0$$

Now, we have thread A stopping to run, and B starting to run. According to (3) R_A will start decaying while R_B increases. When the two become equal, the scheduler will stop B. How much time, t_s, will that take? We already found a formula for that, (9):

$$t_s = \tau ln(1 + (R_A - R_B)/\tau)$$

I.e.,

$$t_s = \tau ln2 \simeq 0.7\,\tau$$

So despite having slept for a very long time, thread B will not get to run now for a very long time - it only accrued enough credit to run for less than τ seconds. As an example, if we have $\tau = 20ms$, we get $t_s \simeq 14ms$. So a thread cannot hope to gain more than a 14ms advantage on its competitors by sleeping for a long time.

Let's look at another aspect of what k means: If thread A didn't run for very long, but rather ran for only $T \ll \tau$, how much will then B allowed to run until the runtime equalizes? Let's again use $\tau = 20ms$.

We'll first try $T = 5ms$. According to the above formulas, after T we have $R_A = 4.42ms$, $R_B = 0$, and so $t_s = \tau ln(1 + (R_A - R_B)/\tau) = 4ms$. So using $\tau = 20ms$ we will not be completely fair even for 5ms durations.

For $T = 1ms$, we get $R_A = 0.97ms$, $R_B = 0$, and so $t_s = \tau ln(1 + (R_A - R_B)/\tau) = 0.95ms$.

The meaning of "p"

As we already explained above, our p measures the thread's *priority* compared to other threads. Perhaps counter-intuitively, the thread with the **lower** p gets more runtime. If one thread has priority p, and the second p/2, the second thread will get twice the runtime as the first.

Generally, the allowed range of p is (0,inf]. the edge-case p=inf is a useful value, which we use for the *idle thread*: A thread with infinite p will, upon running for any period of time accrue an infinite runtime, and from now on will always be last on the run queue, and only run when no other thread wants to run.

Using a very large, but not infinite, p, is not a good idea because R can be up to p/k so that R' can be $c \cdot p/k$ which for very large p, can cause R' to overflow despite precautions we take (i.e., renormalization) to avoid c from overflowing.

On Unix, priorities are traditionally specied as “nice values”, integers in [-20, 20]. The default nice value is 0; 20 is the “nicest” thread (gets least runtime); And -20 is the “meanest” thread (gets most runtime). To convert a nice-value n to and from our priority p, let us choose a constant $\alpha < 1$ (e.g., $\alpha = 0.1$), and decide that the nicest thread (n=-20) would get α times the runtime that a default thread gets, and the meanest thread would get $1/\alpha$ times the runtime of a default thread. With this α we get the following formulas for converting n to p, and vice versa:

$$p = \alpha^{n/20}$$
$$n = 20\ (ln(p)/ln(\alpha))$$

Note how indeed for $n = 20$ we get $p = \alpha$, for $n = -20$ we get $p = 1/\alpha$, and for $n = 0$ we get $p = 1$. Conversely, for $p = \alpha$ we get $n = 20$, for $p = 1/\alpha$ we get $n = -20$, and for $p = 1$ we get $n = 0$.

TODO: consider switching the whole document to working with 1/p. This will make things more intuitive to understand: More intuitively, a higher priority will mean more runtime. We allow priority 0 (the idle priority) and not the infinite priority.

New Threads

We start a new thread with R=0 (so also R’=0, R’’=0 regardless of the current c).
This means a new thread will always start with a lower runtime than any of the running threads, and preempt them. We saw in the τ section above that for $\tau = 20ms$, if a busy thread creates the a new thread, the new thread might run for 14ms before the parent gets to run again.