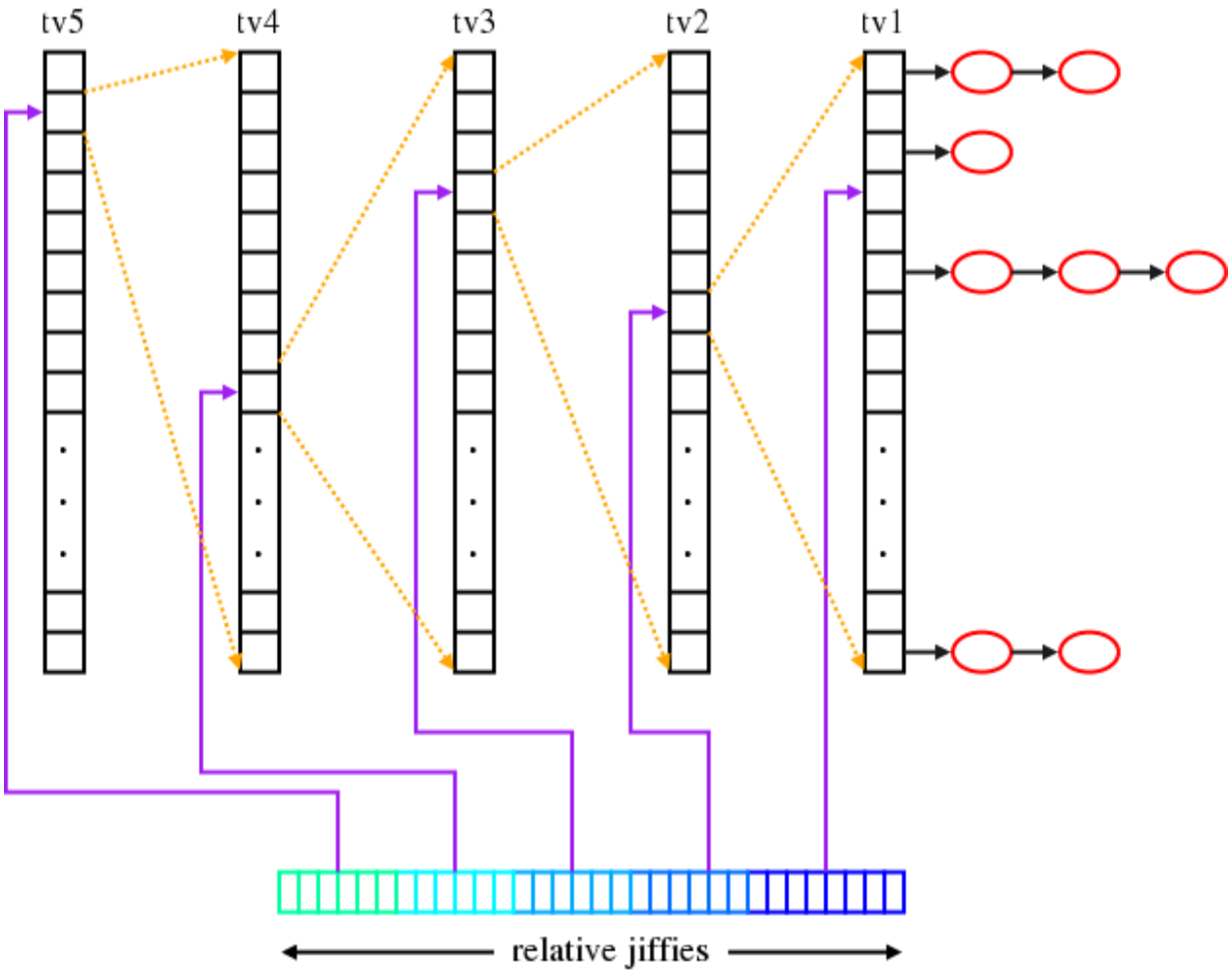
Address: https://lwn.net/Articles/646950/

The kernel's "timer wheel" data structure has served it well for some time; it has changed little since it was described in this article in 2005. There are, however, some shortcomings in its original design that have become more costly over time, and the timer wheel has not adapted well to other changes in the scheduler code. So, after many years, this venerable data structure may soon be replaced with a variant that runs more efficiently, but loses some accuracy in timekeeping in the process.

The kernel maintains two types of timers with two distinct use cases. The high-resolution timer ("hrtimer") mechanism provides accurate timers for work that needs to be done in the near future; hrtimer use is relatively rare, but, when hrtimers are used, they almost always run to completion. "Timeouts," instead, are normally used to alert the kernel to an expected event that has failed to arrive — a missing network packet or I/O completion interrupt, for example. The accuracy requirements for these timers are less stringent (it doesn't matter if an I/O timeout comes a few milliseconds late), and, importantly, these timers are usually canceled before they expire. The timer wheel is used for the latter variety of timers.

Here is the 2005 diagram showing the design of the timer wheel:



This data structure is indexed by the kernel's low-resolution "jiffies" clock; one jiffy corresponds to something between 1ms and 10ms, depending on how the kernel is configured. Once every jiffy, the kernel processes any expired timers. That is done by taking the lowest eight bits of the `jiffies` variable and using them to index into the rightmost array in the above diagram; the result will be a linked list of timer events that expire at the current time.

Every 256 jiffies (in most configurations) the kernel will hit the end of that array; at that point it is necessary to perform a "cascade" operation. Each entry in the next higher array contains 256 jiffies worth of events; the timer code will select the correct entry (by using the next six bits of the `jiffies` value as an index), collect all of the timer entries found there, and distribute them across the 256 entries of the first array according to their expiration times. When the second level is exhausted, it is refilled by cascading down entries from the third level, and so on.

There are a number of advantages to this data structure, including the ability to immediately locate expired entries and quick addition and removal of events. But it has some downsides as well. The cascade operation can be expensive, and the time required is, to a first approximation, unpredictable; that can lead to unwanted latencies elsewhere in the system. The cascade operation is not particularly cache-friendly. There is also no way to quickly determine when the next timer expiration will happen; that requires searching through the wheel to actually find that event. The presence of deferrable timers, which do not have to expire in any sort of timely manner, makes the identification of the next event that actually does have to expire at the requested time harder yet. For these reasons and more, developers have talked about replacing the timer wheel for years.

The new timer wheel

Thomas Gleixner has now posted a first draft of a reinvented timer wheel. It does away with the costly cascade operations (almost all the time) and handles deferrable timers in a much more straightforward manner. These gains come from the realization that not all timers have to be handled with the same level of accuracy.

At a superficial level, the new data structure is quite similar to the old. There is still a hierarchy of arrays containing lists of timer events. In this case, though, the arrays are all the same size (32 entries), and there are eight levels of them. The lowest array contains events with single-jiffy resolution as before, so any new timeout expiring less than 32 jiffies in the future will be placed in this array.

The next array is a little different, though; each entry represents eight jiffies worth of future timer events. Since there are 32 entries in this level as well, it can represent events up to 256 jiffies into the future. Entries in the third level each hold 64 jiffies worth of events; in the fourth level, they hold 512 jiffies worth, and so on. So each level covers a time period eight times longer than the level below it. The numbers are different from the old implementation, but the concept is the same, so far.

The old timer wheel would, each jiffy, run any expiring timers found in the appropriate entry in the highest-resolution array. It would then cascade entries downward if need be, spreading out the timer events among the higher-resolution entries of each lower-level array. The new code also runs the timers from the highest-resolution array in the same way, but the handling of the remaining arrays is different. Every eight jiffies, the appropriate entry in the second-level array will be checked, and any events found there will be expired. The same thing happens in the third level every 64 jiffies, in the fourth level every 512 jiffies, and so on.

The end result is that the timer events in the higher-level arrays will be expired in place, rather than being cascaded downward. This approach, obviously, saves all the effort of performing the cascade. But it also means that any timeout that is more than 31 jiffies in the future will be run with lower accuracy. For example, a timeout that is 36 jiffies in the future will be put in the next higher eight-jiffy slot — 40 jiffies in the future. So that event will expire four jiffies later than requested. As timeouts are placed further into the future, the accuracy of their expiration will decline accordingly. The seventh level in this scheme will hold timeouts that are at least 1,048,576 jiffies in the future with 262,144-jiffy resolution. On a 1000HZ system, that corresponds to timeouts at least 17 minutes in the future; they will expire with a resolution of four minutes.

The old implementation was not subject to this loss of accuracy; even timeouts days in the future would expire at "exactly" the right time, for a one-jiffy value of "exactly." So one could argue that the replacement timer wheel does not work as well. But, first, one should remember that (1) almost all timeouts are set for the near future, (2) almost all timeouts are canceled before expiration, and (3) timeouts indicate that something went wrong and do not need to be delivered with a high degree of accuracy. So sacrificing some of that accuracy for higher timer-wheel performance would appear to be a good tradeoff.

Thomas's patch also dispenses with the timer slack mechanism. Timer slack allows the expiration of timeouts to be deferred; it is intended to cause timeouts to be executed together and reduce the number of times the system wakes up. The new timer wheel batches things naturally for anything but the shortest of timeouts, so there is arguably no longer a need for a separate "slack" mechanism.

Deferrable timers are a bit different though; they can be deferred indefinitely if need be. They usually correspond to some sort of cleanup work that must be done eventually, but with no particular urgency. If the CPU is running in the tickless mode, those timeouts should be deferred for as long as it takes to avoid interrupting the running application. In Thomas's patch, deferrable timers are stored in a separate, parallel timer wheel; this gets them out of the way and eases the task of figuring out when the next timer interrupt should be scheduled.

The new timer wheel code maintains a bitmap with a bit corresponding to each entry in the timer arrays; if there are timeouts stored in that entry, that bit is set. Finding the first array entry with an outstanding timeout is thus a simple matter of finding the first set bit in the bitmap — a fast operation. Then, since the expiration time of each array entry is known, the time of the next expiring timeout can be calculated without actually needing to look at the timeout entry. Placing deferrable timeouts in their own array makes it easy to simply avoid looking at them when checking for this next expiring timeout, speeding the operation further.

This code is all new and untried; Thomas warns that "it might eat your disk, kill your cat and make your kids miss the bus." That would suggest that it is certainly not considered to be 4.2 material. But, with some time and testing, it could likely be ready for a development cycle shortly after that. Then the kernel will, at last, have a shiny, new, faster timer wheel.