Completely Fair Scheduler (CFS)

- used by Linux prior to kernel v6.6
  - recently replaced by earliest eligible virtual deadline first scheduling (EEVDF)
    - EEVDF aims to improve responsiveness and simplify configuration while keeping the fairness of CFS
- based on WFQ
- designed to be highly efficient and scalable
  - also contains many heuristics to improve performance

Weighted Fair Queueing (WFQ)

- RR with different priorities for processes
  - priorities can be decided by scheduler or set by users
  - time slice allocated is proportional to the priority
    - higher priority gets a longer time slice
- wrinkle: it may not be possible to exactly enforce the time slice
  - timer interrupts may not align with slice
  - process may block before the end of its slice
- solution: keep track of the actual running time of each process and schedule the process that has used the least fraction of its fair share (as determined by its priority)
  - adjust future slice sizes to compensate for difference between time used and fair share

Completely Fair Scheduler (CFS)

- the main idea
  - system tracks accumulated *virtual runtime* for each process
  - next process scheduled is the one with the lowest vruntime
  - uses variable-sized time slices
    - *sched_latency* defines maximum wait time (e.g. 48ms)
      - time slice for a process is sched_latency / number of running processes
    - *min_granularity* defines minimum time slice (e.g. 6ms)
      - prevents overly-short time slices and too-frequent context switches

Weighting Priorities

- processes are assigned a *nice* level
  - ranges from -20 (high priority) to +19 (low priority)
    - larger values = more nice = other processes get to run instead of you
  - default 0
- using nice values
  - nice value is mapped to a weight
  - weight is used to compute time slice
    - the weights are defined so that the same difference in nice level results in the same ratio of CPU times
    - a nice level one smaller (higher priority) means 1.25x the time slice
  - weight is also used in the computation of vruntime
    - scales actual runtime to normalize across different sets of processes

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

$$\texttt{time\_slice}_k = \frac{\texttt{weight}_k}{\sum_{i=0}^{n-1} \texttt{weight}_i} \cdot \texttt{sched\_latency}$$

$$\texttt{vruntime}_i = \texttt{vruntime}_i + \frac{\texttt{weight}_0}{\texttt{weight}_i} \cdot \texttt{runtime}_i$$

Dealing With I/O

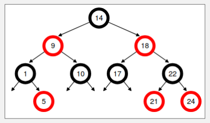
- problem: a process blocked for I/O will not accumulate vruntime
  - when it becomes runnable again, it will monopolize the CPU because it has a lower vruntime than anything else
- solution: set vruntime for newly-ready process to the minimum value in the tree
  - avoids starvation at the expense of I/O-bound processes not getting their fair share of the CPU

CFS Summary

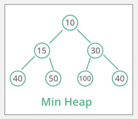
- highly tuned for efficiency with large number of processes
- can handle multi-CPU scheduling
- can handle scheduling process groups
- in principle ensures fair allocation for CPU bound processes over sched_latency interval if there aren't too many processes
  - close to fair if too many processes
  - tracks vruntime exactly to accommodate allocated time slice not aligning exactly with timer interrupts
- I/O bound processes may not get their fair share of CPU time

Algorithmic Notes – Scheduling

- CFS uses a balanced BST to store running/runnable processes in order by vruntime
  - specifically a red-black tree
  - O(log n) insert, remove
  - O(1) find min (if cached)



- this is a priority queue data structure, why not a heap?
  - a heap is a complete binary tree with an ordering property
    - for every node, its key is less than or equal to those of its children (min heap)
    - supports O(1) find min, O(log n) insert, remove min
  - heaps are typically implemented using arrays rather than a linked structure
    - requires a contiguous chunk of memory
    - doesn't scale well with thousands of processes



https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-lottery.pdf
https://www.geeksforgeeks.org/dsa/whats-the-relationship-between-a-heap-and-the-heap/

Algorithmic Notes – Scheduling

- randomization is a powerful technique in a variety of domains
  - effectively dodges worst cases
    - every input is equally likely to be the worst case, and the worst case performance is probabilistically unlikely
  - often a much simpler algorithm and can avoid expensive bookkeeping
    - no need to guarantee worst case performance